Kummer strikes back: new DH speed records

Daniel J. Bernstein[1,2], Chitchanok Chuengsatiansup[2], Tanja Lange[2], and Peter Schwabe[3]

[1] Department of Computer Science, University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
c.chuengsatiansup@tue.nl, tanja@hyperelliptic.org
[3] Radboud University Nijmegen, Digital Security Group
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
peter@cryptojedi.org

Abstract. This paper sets new speed records for high-security constant-time variable-base-point Diffie–Hellman software: 305395 Cortex-A8-slow cycles; 273349 Cortex-A8-fast cycles; 88916 Sandy Bridge cycles; 88448 Ivy Bridge cycles; 54389 Haswell cycles. There are no higher speeds in the literature for any of these platforms.

The new speeds rely on a synergy between (1) state-of-the-art formulas for genus-2 hyperelliptic curves and (2) a modern trend towards vectorization in CPUs. The paper introduces several new techniques for efficient vectorization of Kummer-surface computations.

Keywords: performance, Diffie–Hellman, hyperelliptic curves, Kummer surfaces, vectorization

1 Introduction

The Eurocrypt 2013 paper "Fast cryptography in genus 2" by Bos, Costello, Hisil, and Lauter [17] reported 117000 cycles on Intel's Ivy Bridge microarchitecture for high-security constant-time scalar multiplication on a genus-2 Kummer surface. The eBACS site for publicly verifiable benchmarks [13] confirms 119032 "cycles to compute a shared secret" (quartiles: 118904 and 119232) for the `kumfp127g` software from [17] measured on a single core of `h9ivy`, a 2012 Intel Core i5-3210M running at 2.5GHz. The software is not much slower on Intel's previous microarchitecture, Sandy Bridge: eBACS reports 122716 cycles (quartiles: 122576 and 122836) for `kumfp127g` on `h6sandy`, a 2011 Intel Core i3-2310M running at 2.1GHz. (The quartiles demonstrate that rounding to a

This work was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) under grants 639.073.005, 613.001.011, and through the Veni 2013 project 13114. Permanent ID of this document: `1c5c0ead2524267af6b4f6d9114f10f0`. Date: 2014.10.28.

multiple of 1000 cycles, as in [17], loses statistically significant information; we follow eBACS in reporting medians of exact cycle counts.)

The paper reported that this was a "new software speed record" ("breaking the 120k cycle barrier") compared to "all previous genus 1 and genus 2 implementations" of high-security constant-time scalar multiplication. Obviously the genus-2 cycle counts shown above are better than the (unverified) claim of 137000 Sandy Bridge cycles by Longa and Sica in [40] (Asiacrypt 2012) for constant-time elliptic-curve scalar multiplication; the (unverified) claim of 153000 Sandy Bridge cycles by Hamburg in [34] for constant-time elliptic-curve scalar multiplication; the 182708 cycles reported by eBACS on `h9ivy` for `curve25519`, a constant-time implementation by Bernstein, Duif, Lange, Schwabe, and Yang [11] (CHES 2011) of Bernstein's Curve25519 elliptic curve [9]; and the 194036 cycles reported by eBACS on `h6sandy` for `curve25519`.

One might conclude from these figures that genus-2 hyperelliptic-curve cryptography (HECC) solidly outperforms elliptic-curve cryptography (ECC). However, two newer papers claim better speeds for ECC, and a closer look reveals a strong argument that HECC should have trouble competing with ECC.

The first paper, [44] by Oliveira, López, Aranha, and Rodríguez-Henríquez (CHES 2013 best-paper award), is the new speed leader in eBACS for *non-constant-time* scalar multiplication; the paper reports a new Sandy Bridge speed record of 69500 cycles. Much more interesting for us is that the paper claims 114800 Sandy Bridge cycles for *constant-time* scalar multiplication, beating [17]. eBACS reports 119904 cycles, but this is still faster than [17].

The second paper, [24] by Faz-Hernández, Longa, and Sánchez, claims 92000 Ivy Bridge cycles or 96000 Sandy Bridge cycles for constant-time scalar multiplication; a July 2014 update of the paper claims 89000 Ivy Bridge cycles or 92000 Sandy Bridge cycles. These claims are not publicly verifiable, but if they are even close to correct then they are faster than [17].

Both of these new papers, like [40], rely heavily on curve endomorphisms to eliminate many doublings, as proposed by Gallant, Lambert, and Vanstone [27] (Crypto 2001), patented by the same authors in [28] and [29], and expanded by Galbraith, Lin, and Scott [26] (Eurocrypt 2009). Specifically, [44] uses a GLS curve over a binary field to eliminate 50% of the doublings, while also taking advantage of Intel's new `pclmulqdq` instruction to multiply binary polynomials; [24] uses a GLV+GLS curve over a prime field to eliminate 75% of the doublings.

One can also use the GLV and GLS ideas in genus 2, as explored by Bos, Costello, Hisil, and Lauter starting in [17] and continuing in [18] (CHES 2013). However, the best GLV/GLS speed reported in [18], 92000 Ivy Bridge cycles, provides only 2^{105} security and is not constant time. This is less impressive than the 119032 cycles from [17] for constant-time DH at a 2^{125} security level, and less impressive than the reports in [44] and [24].

The underlying problem for HECC is easy to explain. All known HECC addition formulas are considerably slower than the state-of-the-art ECC addition formulas at the same security level. Almost all of the HECC options explored in

[17] are bottlenecked by additions, so they were doomed from the outset, clearly incapable of beating ECC.

The one exception is that HECC provides an extremely fast *ladder* (see Section 2), built from extremely fast *differential* additions and doublings, considerably faster than the Montgomery ladder frequently used for ECC. This is why [17] was able to set DH speed records.

Unfortunately, differential additions do not allow arbitrary addition chains. Differential additions are incompatible with standard techniques for removing most or all doublings from fixed-base-point single-scalar multiplication, and with standard techniques for removing many doublings from multi-scalar multiplication. As a consequence, differential additions are incompatible with the GLV+GLS approach mentioned above for removing many doublings from single-scalar multiplication. This is why the DH speeds from [17] were quickly superseded by DH speeds using GLV+GLS. A recent paper [22] (Eurocrypt 2014) by Costello, Hisil, and Smith shows feasibility of combining differential additions and use of endomorphisms but reports 145000 Ivy Bridge cycles for constant-time software, much slower than the papers mentioned above.

1.1. Contributions of this paper. We show that HECC has an important compensating advantage, and we exploit this advantage to achieve new DH speed records. The advantage is that we are able to heavily *vectorize* the HECC ladder.

CPUs are evolving towards larger and larger vector units. A low-cost low-power ARM Cortex-A8 CPU core contains a 128-bit vector unit that every two cycles can compute two vector additions, each producing four sums of 32-bit integers, or one vector multiply-add, producing two results of the form $ab + c$ where a, b are 32-bit integers and c is a 64-bit integer. Every cycle a Sandy Bridge CPU core can compute a 256-bit vector floating-point addition, producing four double-precision sums, and at the same time a 256-bit vector floating-point multiplication, producing four double-precision products. A new Intel Haswell CPU core can carry out two 256-bit vector multiply-add instructions every cycle. Intel has announced future support for 512-bit vectors ("AVX-512").

Vectorization has an obvious attraction for a chip manufacturer: the costs of decoding an instruction are amortized across many arithmetic operations. The challenge for the algorithm designer is to efficiently vectorize higher-level computations so that the available circuitry is performing useful work during these computations rather than sitting idle. What we show here is how to fit HECC with surprisingly small overhead into commonly available vector units. This poses several algorithmic challenges, notably to minimize the permutations required for the Hadamard transform (see Section 4). We claim broad applicability of our techniques to modern CPUs, and to illustrate this we analyze all three of the microarchitectures mentioned in the previous paragraph.

Beware that different microarchitectures often have quite different performance. A paper that advertises a "better" algorithmic idea by reporting new record cycle counts on a new microarchitecture, not considered in the previous literature, might actually be reporting an idea that *loses* performance on *all* microarchitectures. We instead emphasize HECC performance on the widely

deployed Sandy Bridge microarchitecture, since Sandy Bridge was shared as a target by the recent ECC speed-record papers listed above. We have now set a new Sandy Bridge DH speed record, demonstrating the value of vectorized HECC. We have also set DH speed records for Ivy Bridge, Haswell, and Cortex-A8.

1.2. Constant time: importance and difficulty. Before stating our performance results we emphasize that our software is truly constant time: the time that we use to compute nP is the same for every 251-bit scalar n and every point P. We strictly follow the rules stated by Bernstein in [9] (PKC 2006): we avoid "all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings". The importance of these data-flow requirements should be clear from, e.g., the Tromer–Osvik–Shamir attack [49] (J. Cryptology 2010) recovering disk-encryption keys from the Linux kernel via cache timings, the Brumley–Tuveri attack [19] (ESORICS 2011) recovering ECDSA keys from OpenSSL via branch timings, and the recent "Lucky Thirteen" AlFardan–Paterson attack [4] (S&P 2013) recovering HTTPS plaintext via decryption timings.

Unfortunately, many of the speed reports in the literature are for cryptographic software that does not meet the same requirements. Sometimes the software is clearly labeled as taking variable time (for example, the ECC speed records from [44] state this quite explicitly), so it is arguably the user's fault for deploying the software in applications that handle secret data; but in other cases non-constant-time software is incorrectly advertised as "constant time".

Consider, for example, the scalar-multiplication algorithm stated in [17, Algorithm 7], which includes a conditional branch for each bit of the scalar n. The "Side-channel resistance" section of the paper states "The main branch, i.e. checking if the bit is set (or not), can be converted into straight-line code by masking (pointers to) the in- and output. Since no lookup tables are used, and all modern cache sizes are large enough to hold the intermediate values ... the algorithm (and runtime) becomes independent of input almost for free."

Unfortunately, the argument in [17] regarding cache sizes is erroneous, and this pointer-swapping strategy does not actually produce constant-time software. An operating-system interrupt can occur at any moment (for example, triggered by a network packet), knocking some or all data out of the cache (presumably at addresses predictable to, or controllable by, an attacker—it is helpful for the attacker that, for cost reasons, cache associativity is limited); see also the "flush+reload" attack from [51] and other local-spy attacks cited in [51]. If P_0 is knocked out of cache and the algorithm accesses P_0 then it suffers a cache miss; if both P_0 and P_1 are subsequently knocked out of cache and the algorithm accesses P_1, P_0 then it suffers two more cache misses. If, on the other hand, P_0 is knocked out of cache and the algorithm accesses P_1 then it does not suffer a cache miss; if both P_0 and P_1 are subsequently knocked out of cache and the algorithm accesses P_0, P_1 then it suffers two more cache misses. The total number of cache misses distinguishes these two examples, revealing whether the algorithm accessed P_0, P_1, P_0 or P_1, P_0, P_1.

We checked the `kumfp127g` software from [17], and found that it contained exactly the branch indicated in [17, Algorithm 7]. This exposes the software not just to data-cache-timing attacks but also to instruction-cache-timing attacks, branch-timing attacks, etc.; for background see, e.g., [3] (CHES 2010). Evidently "can be converted" was a statement regarding possibilities, not a statement regarding what was actually done in the benchmarked software.

This is not an isolated example. We checked the fastest "constant-time" software from [22] and found that it contained key-dependent branches. Specifically, the secret key `sk` is passed to a function `mon_fp_smul_2e127m1e2_djb`, which calls `decompose` and then `getchain` to convert the secret key into a `scanned` array and then calls `ec_fp_mdbladdadd_2e127m1e2_asm` repeatedly using various secret bits from that array; `ec_fp_mdbladdadd_2e127m1e2_asm` uses "`jmpq *(%rsi)`" to branch to different instructions depending on those bits. This exposes the software to instruction-cache-timing attacks.

The correct response to timing attacks is to use constant-time arithmetic instructions to simulate data-dependent branches, data-dependent table indices, etc.; see, e.g., Section 4.5. It is essential for "constant-time" cryptographic software to go to this effort. The time required for this simulation is often highly algorithm-dependent, and must be included in speed reports so that users are not misled regarding the costs of security.

Of course, the security assessment above was aided by the availability of the source code from [17] and [22]. For comparison, the public has no easy way to check the "constant time" claims for the software in [24], so for users the only safe assumption is that the claims are not correct. If that software is deployed somewhere then an attacker can be expected to do the necessary reverse-engineering work to discover and exploit the timing variability.

Our comparisons below are limited to software that has been advertised in the literature to be constant-time. Some of this software is not actually constant-time, as illustrated by the analysis above, and would become slower if it were fixed.

The authors of [17] and [22] have now updated their software, with credit to us. Their new (slower) software is not yet integrated into eBACS; our comparison table below shows the older software.

1.3. Performance results. eBACS shows that on a single core of `h6sandy` our DH software ("`kummer`") uses just 88916 Sandy Bridge cycles (quartiles: 88868 and 89184). On a single core of `h9ivy` our software uses 88448 cycles (quartiles: 88424 and 88476).

On a single core of `titan0`, an Intel Xeon E3-1275 V3 (Haswell), our software uses 54389 cycles (quartiles: 54341 and 54454). See Section 1.4 for previous Haswell results.

For Cortex-A8 there is a difference in L1-cache performance between Cortex-A8-"fast" CPUs and Cortex-A8-"slow" CPUs. On `h7beagle`, a TI Sitara AM3359 (Cortex-A8-slow), our software uses 305395 cycles (quartiles: 305380 and 305413). On `h4mx515e`, a Freescale i.MX515 (Cortex-A8-fast), our software uses 273349

arch	cycles	ladder	open	g	field	source of software
A8-slow	497389	yes	yes	1	$2^{255}-19$	[15] CHES 2012
A8-slow	305395	yes	yes	2	$2^{127}-1$	**new (this paper)**
A8-fast	460200	yes	yes	1	$2^{255}-19$	[15] CHES 2012
A8-fast	273349	yes	yes	2	$2^{127}-1$	**new (this paper)**
Sandy	194036	yes	yes	1	$2^{255}-19$	[11] CHES 2011
Sandy	153000?	yes	no	1	$2^{252}-2^{232}-1$	[34]
Sandy	137000?	no	no	1	$(2^{127}-5997)^2$	[40] Asiacrypt 2012
Sandy	122716	yes	yes	2	$2^{127}-1$	[17] Eurocrypt 2013
Sandy	119904	no	yes	1	2^{254}	[44] CHES 2013
Sandy	96000?	no	no	1	$(2^{127}-5997)^2$	[24] CT-RSA 2014
Sandy	92000?	no	no	1	$(2^{127}-5997)^2$	[24] July 2014
Sandy	88916	yes	yes	2	$2^{127}-1$	**new (this paper)**
Ivy	182708	yes	yes	1	$2^{255}-19$	[11] CHES 2011
Ivy	145000?	yes	yes	1	$(2^{127}-1)^2$	[22] Eurocrypt 2014
Ivy	119032	yes	yes	2	$2^{127}-1$	[17] Eurocrypt 2013
Ivy	114036	no	yes	1	2^{254}	[44] CHES 2013
Ivy	92000?	no	no	1	$(2^{127}-5997)^2$	[24] CT-RSA 2014
Ivy	89000?	no	no	1	$(2^{127}-5997)^2$	[24] July 2014
Ivy	88448	yes	yes	2	$2^{127}-1$	**new (this paper)**
Haswell	145907	yes	yes	1	$2^{255}-19$	[11] CHES 2011
Haswell	100895	yes	yes	2	$2^{127}-1$	[17] Eurocrypt 2013
Haswell	55595	no	yes	1	2^{254}	[44] CHES 2013
Haswell	54389	yes	yes	2	$2^{127}-1$	**new (this paper)**

Table 1.5. Reported high-security DH speeds for Cortex-A8, Sandy Bridge, Ivy Bridge, and Haswell. Cycle counts from eBACS are for `curve25519`, `kumfp127g`, `gls254prot`, and our `kummer` on `h7beagle` (Cortex-A8-slow), `h4mx515e` (Cortex-A8-fast), `h6sandy` (Sandy Bridge), `h9ivy` (Ivy Bridge), and `titan0` (Haswell). Cycle counts not from SUPERCOP are marked "?". ECC has $g = 1$; genus-2 HECC has $g = 2$. See text for security requirements.

cycles (quartiles: 273337 and 273387). See Section 1.4 for previous Cortex-A8 results.

These cycle counts are the complete time for constant-time variable-scalar variable-base-point single-scalar multiplication using SUPERCOP's `crypto_dh` API. Our inputs and outputs are canonical representations of points as 48-byte strings and scalars as 32-byte strings. Our timings include more than just scalar multiplication on an internal representation of field elements; they also include the costs of parsing strings, all other necessary setup, the costs of conversion to inverted-affine form $(x/y, x/z, x/t)$ in the notation of Section 2, the costs of converting lazily reduced field elements to unique representatives, and the costs of converting to strings.

1.4. Cycle-count comparison. Table 1.5 summarizes reported high-security DH speeds for Cortex-A8, Sandy Bridge, Ivy Bridge, and Haswell.

This table is limited to software that *claims* to be constant time, and that claims a security level close to 2^{128}. This is the reason that the table does not include, e.g., the 767000 Cortex-A8 cycles and 108000 Ivy Bridge cycles claimed in [18] for constant-time scalar multiplication on a Kummer surface; the authors claim only 103 bits of security for that surface. This is also the reason that the table does not include, e.g., the 69500 Sandy Bridge cycles claimed in [44] for non-constant-time scalar multiplication.

The table does not attempt to report whether the listed cycle counts are from software that actually meets the above security requirements. In some cases inspection of the software has shown that the security requirements are violated; see Section 1.2. "Open" means that the software is reported to be open source, allowing third-party inspection.

Our speeds, on the same platform targeted in [17], solidly beat the HECC speeds from [17]. Our speeds also solidly beat the Cortex-A8, Sandy Bridge, and Ivy Bridge speeds from all available ECC software, including [11], [15], [22], and [44]; solidly beat the speeds claimed in [34] and [40]; and are even faster than the July 2014 Sandy Bridge/Ivy Bridge DH record claimed in [24], namely 92000/89000 cycles using unpublished software for GLV+GLS ECC. For Haswell, despite Haswell's exceptionally fast binary-field multiplier, our speeds beat the 55595 cycles from [44] for a GLS curve over a binary field. We set our new speed records using an HECC ladder that is conceptually much simpler than GLV and GLS, avoiding all the complications of scalar-dependent precomputations, lattice size issues, multi-scalar addition chains, endomorphism-rho security analysis, Weil-descent security analysis, and patents.

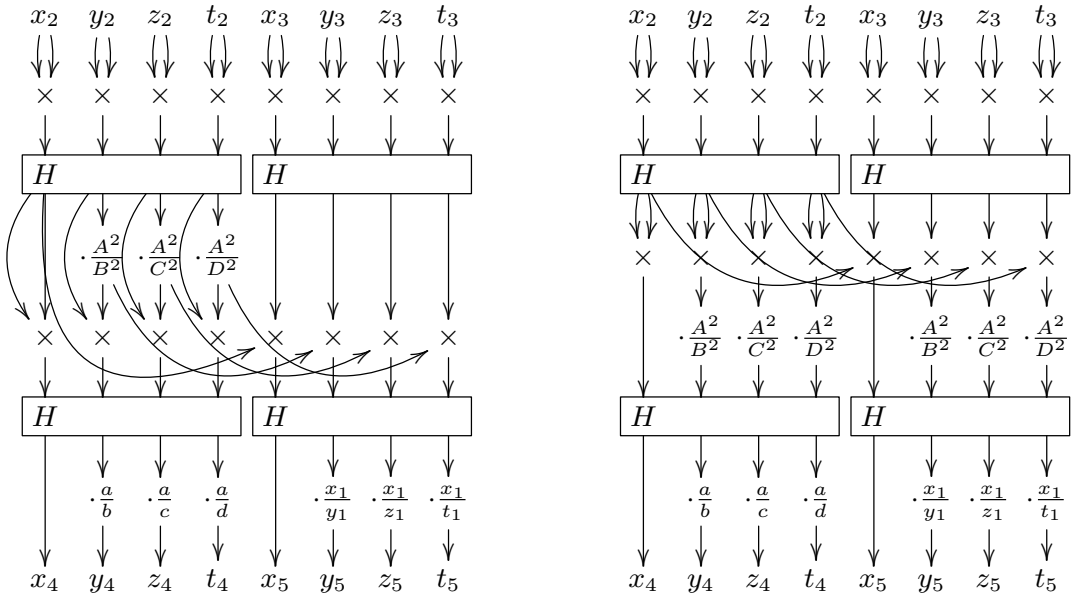
2 Fast scalar multiplication on the Kummer surface

This section reviews the smallest number of field operations known for genus-2 scalar multiplication. Sections 3 and 4 optimize the performance of those field operations using 4-way vector instructions.

Vectorization changes the interface between this section and subsequent sections. What we actually optimize is not individual field operations, but rather pairs of operations, pairs of pairs, etc., depending on the amount of vectorization available from the CPU. Our optimization also takes advantage of sequences of operations such as the output of a squaring being multiplied by a small constant. What matters in this section is therefore not merely the *number* of field multiplications, squarings, etc., but also the *pattern* of those operations.

2.1. Only 25 multiplications. Almost thirty years ago Chudnovsky and Chudnovsky wrote a classic paper [21] optimizing scalar multiplication inside the elliptic-curve method of integer factorization. At the end of the paper they also considered the performance of scalar multiplication on Jacobian varieties of genus-2 hyperelliptic curves. After mentioning various options they gave some details of one option, namely scalar multiplication on a Kummer surface.

A Kummer surface is related to the Jacobian of a genus-2 hyperelliptic curve in the same way that x-coordinates are related to a Weierstrass elliptic curve. There



(a) $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas. (b) $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas.

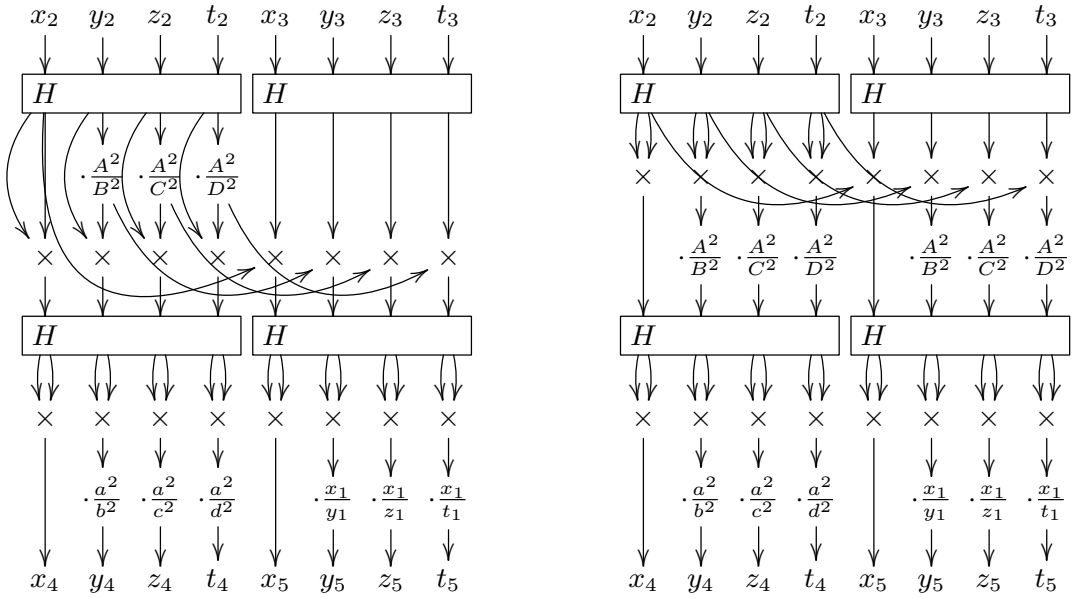
Fig. 2.2. Ladder formulas for the Kummer surface. Inputs are $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$, $X(P) = (x_2 : y_2 : z_2 : t_2)$, and $X(Q) = (x_3 : y_3 : z_3 : t_3)$; outputs are $X(2P) = (x_4 : y_4 : z_4 : t_4)$ and $X(P + Q) = (x_5 : y_5 : z_5 : t_5)$. Formulas in (a) are from Gaudry [30]; diagrams are copied from Bernstein [10].

is a standard rational map X from the Jacobian to the Kummer surface; this map satisfies $X(P) = X(-P)$ for points P on the Jacobian and is almost everywhere exactly 2-to-1. Addition on the Jacobian does not induce an operation on the Kummer surface (unless the number of points on the surface is extremely small), but scalar multiplication $P \mapsto nP$ on the Jacobian induces scalar multiplication $X(P) \mapsto X(nP)$ on the Kummer surface. Not every genus-2 hyperelliptic curve can have its Jacobian mapped to the standard type of Kummer surface over the base field, but a noticeable fraction of curves can; see [31].

Chudnovsky and Chudnovsky reported $14\mathbf{M}$ for doubling a Kummer-surface point, where $\mathbf{M}$ is the cost of field multiplication; and $23\mathbf{M}$ for "general addition", presumably differential addition, computing $X(Q+P)$ given $X(P), X(Q)$, $X(Q-P)$. They presented their formulas for doubling, commenting on a "pretty symmetry" in the formulas and on the number of multiplications that were actually squarings. They did not present their formulas for differential addition.

Two decades later, in [30], Gaudry reduced the total cost of differential addition and doubling, computing $X(2P), X(Q + P)$ given $X(P), X(Q), X(Q - P)$, to $25\mathbf{M}$, more precisely $16\mathbf{M} + 9\mathbf{S}$, more precisely $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$, where $\mathbf{S}$ is the cost of field squaring and $\mathbf{m}$ is the cost of multiplication by a curve constant. An ℓ-bit scalar-multiplication ladder therefore costs just $10\ell\mathbf{M} + 9\ell\mathbf{S} + 6\ell\mathbf{m}$.

Gaudry's formulas are shown in Figure 2.2(a). Each point on the Kummer surface is expressed projectively as four field elements $(x : y : z : t)$; one is free to replace $(x : y : z : t)$ with $(rx : ry : rz : rt)$ for any nonzero r. The "H"



(a) $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ ladder formulas. (b) $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$ ladder formulas.

Fig. 2.4. Ladder formulas for the squared Kummer surface. Compare to Figure 2.2.

boxes are Hadamard transforms, each using 4 additions and 4 subtractions; see Section 4. The Kummer surface is parametrized by various constants $(a : b : c : d)$ and related constants $(A^2 : B^2 : C^2 : D^2) = H(a^2 : b^2 : c^2 : d^2)$. The doubling part of the diagram, from $(x_2 : y_2 : z_2 : t_2)$ down to $(x_4 : y_4 : z_4 : t_4)$, uses $3\mathbf{M} + 5\mathbf{S} + 6\mathbf{m}$, matching the $14\mathbf{M}$ reported by Chudnovsky and Chudnovsky; but the rest of the picture uses just $7\mathbf{M} + 4\mathbf{S}$ extra, making remarkable reuse of the intermediate results of doubling. Figure 2.2(b) replaces $10\mathbf{M} + 9\mathbf{S} + 6\mathbf{m}$ with $7\mathbf{M} + 12\mathbf{S} + 9\mathbf{m}$, as suggested by Bernstein in [10]; this saves time if $\mathbf{m}$ is smaller than the difference $\mathbf{M} - \mathbf{S}$.

2.3. The original Kummer surface vs. the squared Kummer surface. Chudnovsky and Chudnovsky had actually used slightly different formulas for a slightly different surface, which we call the "squared Kummer surface". Each point $(x : y : z : t)$ on the original Kummer surface corresponds to a point $(x^2 : y^2 : z^2 : t^2)$ on the squared Kummer surface. Figure 2.4 presents the equivalent of Gaudry's formulas for the squared Kummer surface, relabeling $(x^2 : y^2 : z^2 : t^2)$ as $(x : y : z : t)$; the squarings at the top of Figure 2.2 have moved close to the bottom of Figure 2.4.

The number of field operations is the same either way, as stated in [10] with credit to André Augustyniak. However, the squared Kummer surface has a computational advantage over the original Kummer surface, as pointed out by Bernstein in [10]: constructing surfaces in which all of $a^2, b^2, c^2, d^2, A^2, B^2, C^2, D^2$ are small, producing fast multiplications by constants in Figure 2.4, is easier than constructing surfaces in which all of $a, b, c, d, A^2, B^2, C^2, D^2$ are small, producing fast multiplications by constants in Figure 2.2.

2.5. Preliminary comparison to ECC. A Montgomery ladder step for ECC costs $5\mathbf{M}+4\mathbf{S}+1\mathbf{m}$, while a ladder step on the Kummer surface costs $10\mathbf{M}+9\mathbf{S}+6\mathbf{m}$ or $7\mathbf{M}+12\mathbf{S}+9\mathbf{m}$. Evidently ECC uses only about half as many operations. However, for security ECC needs primes around 256 bits (such as the convenient prime $2^{255}-19$), while the Kummer surface can use primes around 128 bits (such as the even more convenient prime $2^{127}-1$), and presumably this saves more than a factor of 2.

Several years ago, in [10], Bernstein introduced 32-bit Intel Pentium M software for generic Kummer surfaces (i.e., $\mathbf{m}=\mathbf{M}$) taking about 10% fewer cycles than his Curve25519 software, which at the time was the speed leader for ECC. Gaudry, Houtmann, and Thomé, as reported in [32, comparison table], introduced 64-bit software for Curve25519 and for a Kummer surface; the second option was slightly faster on AMD Opteron K8 but the first option was slightly faster on Intel Core 2. It is not at all clear that one can reasonably extrapolate to today's CPUs.

Bernstein's cost analysis concluded that HECC could be as much as 1.5× faster than ECC on a Pentium M (cost 1355 vs. cost 1998 in [10, page 31]), depending on the exact size of the constants $a^2, b^2, c^2, d^2, A^2, B^2, C^2, D^2$. This motivated a systematic search through small constants to find a Kummer surface providing high security and high twist security. But this was more easily said than done: genus-2 point counting was much more expensive than elliptic-curve point counting.

2.6. The Gaudry–Schost Kummer surface. Years later, after a 1000000-CPU-hour computation relying on various algorithmic improvements to genus-2 point counting, Gaudry and Schost announced in [33] that they had found a secure Kummer surface $(a^2 : b^2 : c^2 : d^2) = (11 : -22 : -19 : -3)$ over $\mathbf{F}_p$ with $p = 2^{127}-1$, with Jacobian order and twisted Jacobian order equal to

$$16\cdot 180925139433306555357191732647120652144130617439968355857167262354635672633 9,$$

$$16\cdot 180925139433306555341467595505029059892350884363594131307776729780117962605 1$$

respectively. This is exactly the surface that was used for the HECC speed records in [17]. We obtain even better speeds for the same surface.

Note that, as mentioned by Bos, Costello, Hisil, and Lauter in [17], the constants $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2) = (1 : -1/2 : -11/19 : -11/3)$ in Figure 2.4 are projectively the same as $(-114 : 57 : 66 : 418)$. The common factor 11 between $a^2 = 11$ and $b^2 = -22$ helps keep these integers small. The constants $(1 : A^2/B^2 : A^2/C^2 : A^2/D^2) = (1 : -3 : -33/17 : -33/49)$ are projectively the same as $(-833 : 2499 : 1617 : 561)$.

3 Decomposing field multiplication

The only operations in Figures 2.2 and 2.4 are the H boxes, which we analyze in Section 4, and field multiplications, which we analyze in this section. Our

goal here is to obtain the smallest possible number of CPU cycles for **M**, **S**, etc. modulo $p = 2^{127} - 1$.

This prime has been considered before, for example in [8] and [10]. What is new here is fitting arithmetic modulo this prime, for the pattern of operations shown in Figure 2.4, into the vector abilities of modern CPUs. There are four obvious dimensions of vectorizability:

- Vectorizing across the "limbs" that represent a field element such as x_2. The most obvious problem with this approach is that, when f is multiplied by g, each limb of f needs to communicate with each limb of g and each limb of output. A less obvious problem is that the optimal number of limbs is CPU-dependent and is usually nonzero modulo the vector length. Each of these problems poses a challenge in organizing and reshuffling data inside multiplications.
- Vectorizing across the four field elements that represent a point. All of the multiplications in Figure 2.4 are visually organized into 4-way vectors, except that in some cases the vectors have been scaled to create a multiplication by 1. Even without vectorization, most of this scaling is undesirable for any surface with small a^2, b^2, c^2, d^2: e.g., for the Gaudry–Schost surface we replace $(1 : a^2/b^2 : a^2/c^2 : a^2/d^2)$ with $(-114 : 57 : 66 : 418)$. The only remaining exception is the multiplication by 1 in $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ where $X(Q - P) = (x_1 : y_1 : z_1 : t_1)$. Vectorizing across the four field elements means that this multiplication costs $1\mathbf{M}$, increasing the cost of a ladder step from $7\mathbf{M} + 12\mathbf{S} + 12\mathbf{m}$ to $8\mathbf{M} + 12\mathbf{S} + 12\mathbf{m}$.
- Vectorizing between doubling and differential addition. For example, in Figure 2.4(b), squarings are imperfectly paired with multiplications on the third line; multiplications by constants are perfectly paired with multiplications by the same constants on the fourth line; squarings are perfectly paired with squarings on the sixth line; and multiplications by constants are imperfectly paired with multiplications by inputs on the seventh line. There is some loss of efficiency in, e.g., pairing the squaring with the multiplication, since this prohibits using faster squaring methods.
- Vectorizing across a batch of independent scalar-multiplication inputs, in applications where a suitably sized batch is available. This is relatively straightforward but increases cache traffic, often to problematic levels. In this paper we focus on the traditional case of a single input.

The second dimension of vectorizability is, as far as we know, a unique feature of HECC, and one that we heavily exploit for high performance.

For comparison, one can try to vectorize the well-known Montgomery ladder for ECC [42] across the field elements that represent a point, but (1) this provides only two-way vectorization (x and z), not four-way vectorization; and (2) many of the resulting pairings are imperfect. The Montgomery ladder for Curve25519 was vectorized by Costigan and Schwabe in [23] for the Cell, and then by Bernstein and Schwabe in [15] for the Cortex-A8, but both of those vectorizations had substantially higher overhead than our new vectorization of the HECC ladder.

3.1. Sandy Bridge floating-point units. The only fast multiplier available on Intel's 32-bit platforms for many years, from the original Pentium twenty years ago through the Pentium M, was the floating-point multiplier. This was exploited by Bernstein for cryptographic computations in [8], [9], etc.

The conventional wisdom is that this use of floating-point arithmetic was rendered obsolete by the advent of 64-bit platforms: in particular, Intel now provides a reasonably fast 64-bit integer multiplier. However, floating-point units have also become more powerful; evidently Intel sees many applications that rely critically upon fast floating-point arithmetic. We therefore revisit Bernstein's approach, with the added challenge of vectorization.

We next describe the relevant features of the Sandy Bridge; see [25] for more information. Our optimization of HECC for the Sandy Bridge occupies the rest of Sections 3 and 4. The Ivy Bridge has the same features and should be expected to produce essentially identical performance for this type of code. The Haswell has important differences and is analyzed in Appendix B; the Cortex-A8 is analyzed in Section 5.

Each Sandy Bridge core has several 256-bit vector units operating in parallel on vectors of 4 double-precision floating-point numbers:

- "Port 0" handles one vector multiplication each cycle, with latency 5.
- Port 1 handles one vector addition each cycle, with latency 3.
- Port 5 handles one permutation instruction each cycle. The selection of permutation instructions is limited and is analyzed in detail in Section 4.
- Ports 2, 3, and 4 handle vector loads and stores, with latency 4 from L1 cache and latency 3 to L1 cache. Load/store throughput is limited in various ways, never exceeding one 256-bit load per cycle.

Recall that a double-precision floating-point number occupies 64 bits, including a sign bit, a power of 2, and a "mantissa". Every integer between -2^{53} and 2^{53} can be represented exactly as a double-precision floating-point number. More generally, every real number of the form $2^e i$, where e is a small integer and i is an integer between -2^{53} and 2^{53}, can be represented exactly as a double-precision floating-point number. The computations discussed here do not approach the lower or upper limits on e, so we do not review the details of the limits.

Our final software uses fewer multiplications than additions, and fewer permutations than multiplications. This does not mean that we were free to use extra multiplications and permutations: if multiplications and permutations are not finished quickly enough then the addition unit will sit idle waiting for input. In many cases, noted below, we have the flexibility to convert multiplications to additions, reducing latency; we found that in some cases this saved time despite the obvious addition bottleneck.

3.2. Optimizing M (field multiplication). We decompose an integer f modulo $2^{127}-1$ into six floating-point limbs in (non-integer) radix $2^{127/6}$. This means that we write f as $f_0+f_1+f_2+f_3+f_4+f_5$ where f_0 is a small multiple of 2^0, f_1 is a small multiple of 2^{22}, f_2 is a small multiple of 2^{43}, f_3 is a small multiple of 2^{64}, f_4 is a small multiple of 2^{85}, and f_5 is a small multiple of 2^{106}. (The exact

meaning of "small" is defined by a rather tedious, but verifiable, collection of bounds on the floating-point numbers appearing in each step of the program. It should be obvious that a simpler definition of "small" would compromise efficiency; for example, H cannot be efficient unless the bounds on H intermediate results and outputs are allowed to be larger than the bounds on H inputs.)

If g is another integer similarly decomposed as $g_0 + g_1 + g_2 + g_3 + g_4 + g_5$ then f_0g_0 is a multiple of 2^0, $f_0g_1 + f_1g_0$ is a multiple of 2^{22}, $f_0g_2 + f_1g_1 + f_2g_0$ is a multiple of 2^{43}, etc. Each of these sums is small enough to fit exactly in a double-precision floating-point number, and the total of these sums is exactly fg. What we actually compute are the sums

$$\begin{aligned}
h_0 &= f_0g_0 + 2^{-127}f_1g_5 + 2^{-127}f_2g_4 + 2^{-127}f_3g_3 + 2^{-127}f_4g_2 + 2^{-127}f_5g_1,\\
h_1 &= f_0g_1 + f_1g_0 + 2^{-127}f_2g_5 + 2^{-127}f_3g_4 + 2^{-127}f_4g_3 + 2^{-127}f_5g_2,\\
h_2 &= f_0g_2 + f_1g_1 + f_2g_0 + 2^{-127}f_3g_5 + 2^{-127}f_4g_4 + 2^{-127}f_5g_3,\\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 2^{-127}f_4g_5 + 2^{-127}f_5g_4,\\
h_4 &= f_0g_4 + f_1g_3 + f_2g_2 + f_3g_1 + f_4g_0 + 2^{-127}f_5g_5,\\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0,
\end{aligned}$$

whose total h is congruent to fg modulo $2^{127} - 1$.

There are 36 multiplications f_ig_j here, and 30 additions. (This operation count does not include carries; we analyze carries below.) One can collect the multiplications by 2^{-127} into 5 multiplications such as $2^{-127}(f_4g_5 + f_5g_4)$. We use another approach, precomputing $2^{-127}f_1, 2^{-127}f_2, 2^{-127}f_3, 2^{-127}f_4, 2^{-127}f_5$, for two reasons: first, this reduces the latency of each h_i computation, giving us more flexibility in scheduling; second, this gives us an opportunity to share precomputations when the input f is reused for another multiplication.

3.3. Optimizing S (field squaring) and m (constant field multiplication). For **S**, i.e., for $f = g$, we have

$$\begin{aligned}
h_0 &= f_0f_0 + 2^{-127}2f_1f_5 + 2^{-127}2f_2f_4 + 2^{-127}f_3f_3,\\
h_1 &= 2f_0f_1 + 2^{-127}2f_2f_5 + 2^{-127}2f_3f_4,\\
h_2 &= 2f_0f_2 + f_1f_1 + 2^{-127}2f_3f_5 + 2^{-127}f_4f_4,\\
h_3 &= 2f_0f_3 + 2f_1f_2 + 2^{-127}2f_4f_5,\\
h_4 &= 2f_0f_4 + 2f_1f_3 + f_2f_2 + 2^{-127}f_5f_5,\\
h_5 &= 2f_0f_5 + 2f_1f_4 + 2f_2f_3.
\end{aligned}$$

We precompute $2f_1, 2f_2, 2f_3, 2f_4, 2f_5$ and $2^{-127}f_3, 2^{-127}f_4, 2^{-127}f_5$; this costs 8 multiplications, where 5 of the multiplications can be freely replaced by additions. The rest of **S**, after this precomputation, takes 21 multiplications and 15 additions, plus the cost of carries.

For **m** we have simply $h_0 = cf_0$, $h_1 = cf_1$, etc., costing 6 multiplications plus the cost of carries. This does not work for arbitrary field constants, but it does work for the small constants stated in Section 2.6.

3.4. Carries. The output limbs h_i from $\mathbf{M}$ are too large to be used in a subsequent multiplication. We carry $h_0 \to h_1$ by rounding $2^{-22}h_0$ to an integer c_0, adding $2^{22}c_0$ to h_1, and subtracting $2^{22}c_0$ from h_0. This takes 3 additions (the CPU has a rounding instruction, `vroundpd`, that costs just 1 addition) and 2 multiplications. The resulting h_0 is guaranteed to be between -2^{21} and 2^{21}.

We could similarly carry $h_1 \to h_2 \to h_3 \to h_4 \to h_5$, and carry $h_5 \to h_0$ as follows: round $2^{-127}h_5$ to an integer c_5, add c_5 to h_0, and subtract $2^{127}c_5$ from h_5. One final carry $h_0 \to h_1$, for a total of 7 carries (21 additions and 14 multiplications), would then guarantee that all of $h_0, h_1, h_2, h_3, h_4, h_5$ are small enough to be input to a subsequent multiplication.

The problem with this carry chain is that it has extremely high latency: 5 cycles for $2^{-22}h_0$, 3 more cycles for c_0, 5 more cycles for $2^{22}c_0$, and 3 more cycles to add to h_1, all repeated 7 times, for a total of 112 cycles, plus the latency of obtaining h_0 in the first place. The ladder step in Figure 2.4 has a serial chain of $H \to \mathbf{M} \to \mathbf{m} \to H \to \mathbf{S} \to \mathbf{M}$, for a total latency above 500 cycles, i.e., above 125500 cycles for a 251-bit ladder.

We do better in six ways. First, we use only 6 carries in $\mathbf{M}$ rather than 7, if the output will be used only for $\mathbf{m}$. Even if the output h_0 is several bits larger than 2^{22}, it will not overflow the small-constant multiplication, since our constants are all bounded by 2^{12}.

Second, pushing the same idea further, we do these 6 carries in parallel. First we round in parallel to obtain $c_0, c_1, c_2, c_3, c_4, c_5$, then we subtract in parallel, then we add in parallel, allowing all of $h_0, h_1, h_2, h_3, h_4, h_5$ to end up several bits larger than they would have been with full carries.

Third, we also use 6 parallel carries for a multiplication that *is* an $\mathbf{m}$. There is no need for a chain, since the initial $h_0, h_1, h_2, h_3, h_4, h_5$ cannot be very large.

Fourth, we also use 6 parallel carries for each $\mathbf{S}$. This allows the $\mathbf{S}$ output to be somewhat larger than the input, but this still does not create overflows in the subsequent $\mathbf{M}$. At this point the only remaining block of 7 carries is in the $\mathbf{M^4}$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$, where $\mathbf{M^4}$ means a vector of four field multiplications.

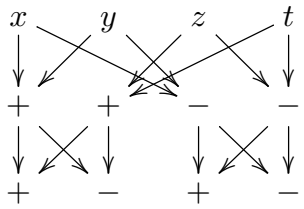
Fifth, for that $\mathbf{M^4}$, we run two carry chains in parallel, carrying $h_0 \to h_1$ and $h_3 \to h_4$, then $h_1 \to h_2$ and $h_4 \to h_5$, then $h_2 \to h_3$ and $h_5 \to h_0$, then $h_3 \to h_4$ and $h_0 \to h_1$. This costs 8 carries rather than 7 but chops latency in half.

Finally, for that $\mathbf{M^4}$, we use the carry approach from [8]: add the constant $\alpha_{22} = 2^{22}(2^{52}+2^{51})$ to h_0, and subtract α_{22} from the result, obtaining the closest multiple of 2^{22} to h_0; add this multiple to h_1 and subtract it from h_0. This costs 4 additions rather than 3, but reduces carry latency from 16 to 9, and also saves two multiplications.

4 Permutations: vectorizing the Hadamard transform

The Hadamard transform H in Section 2 is defined as follows: $H(x, y, z, t) = (x+y+z+t, x+y-z-t, x-y+z-t, x-y-z+t)$. Evaluating this as written would use 12 field additions (counting subtraction as addition), but a standard

"fast Hadamard transform" reduces the 12 to 8:



We copied this diagram from Bernstein [10].

Our representation of field elements for the Sandy Bridge (see Section 3) requires 6 limb additions for each field addition. There is no need to carry before the subsequent multiplications; this is the main reason that we use 6 limbs rather than 5.

In a ladder step there are 4 copies of H, each requiring 8 field additions, each requiring 6 limb additions, for a total of 192 limb additions. This operation count suggests that 48 vector instructions suffice. Sandy Bridge has a helpful `vaddsubpd` instruction that computes $(a-e, b+f, c-g, d+h)$ given (a,b,c,d) and (e,f,g,h), obviously useful inside H.

However, we cannot simply vectorize across x, y, z, t. In Section 3 we were multiplying one x by another, at the same time multiplying one y by another, etc., with no permutations required; in this section we need to add x to y, and this requires permutations.

The Sandy Bridge has a vector permutation unit acting in parallel with the adder and the multiplier, as noted in Section 3. But this does not mean that the cost of permutations can be ignored. A long sequence of permutations inside H will force the adder and the multiplier to remain idle, since only a small fraction of the work inside $\mathbf{M}$ can begin before H is complete.

Our original software used 48 vector additions and 144 vector permutations for the 4 copies of H. We then tackled the challenge of minimizing the number of permutations. We ended up reducing this number from 144 to just 36. This section presents the details; analyzes conditional swaps, which end up consuming further time in the permutation unit; and concludes by analyzing the total number of operations used in our Sandy Bridge software.

4.1. Limitations of the Sandy Bridge permutations. There is a latency-1 permutation instruction `vpermilpd` that computes (y,x,t,z) given (x,y,z,t). `vaddsubpd` then produces $(x-y, y+x, z-t, t+z)$, which for the moment we abbreviate as (e,f,g,h). At this point we seem to be halfway done: the desired output is simply $(f+h, f-h, e+g, e-g)$.

If we had (f,h,e,g) at this point, rather than (e,f,g,h), then we could apply `vpermilpd` and `vaddsubpd` again, obtaining $(f-h, h+f, e-g, g+e)$. One final `vpermilpd` would then produce the desired $(f+h, f-h, e+g, e-g)$. The remaining problem is the middle permutation of (e,f,g,h) into (f,h,e,g).

Unfortunately, Sandy Bridge has very few options for moving data between the left half of a vector, in this case (e,f), and the right half of a vector, in this case (g,h). There is a `vperm2f128` instruction (1-cycle throughput but latency

2) that produces (g, h, e, f), but it cannot even produce (h, g, f, e), never mind a combination such as (f, h, e, g). (Haswell has more permutation instructions, but Ivy Bridge does not. This is not a surprising restriction: n-bit vector units are often designed as $n/2$-bit vector units operating on the left half of a vector in one cycle and the right half in the next cycle, but this means that any communication between left and right requires careful attention in the circuitry. A similar left-right separation is even more obvious for the Cortex-A8.) We could shift some permutation work to the load/store unit, but this would have very little benefit, since simulating a typical permutation requires quite a few loads and stores.

The `vpermilpd` instruction $(x, y, z, t) \mapsto (y, x, t, z)$ mentioned above is one of a family of 16 `vpermilpd` instructions that produce (x or y, x or y, z or t, z or t). There is an even more general family of 16 `vshufpd` instructions that produce (a or b, x or y, c or d, z or t) given (a, b, c, d) and (x, y, z, t). In the first versions of our software we applied `vshufpd` to (e, f, g, h) and (g, h, e, f), obtaining (f, h, g, e), and then applied `vpermilpd` to obtain (f, h, e, g).

Overall a single H handled in this way uses, for each limb, 2 `vaddsubpd` instructions and 6 permutation instructions, half of which are handling the permutation of (e, f, g, h) into (f, h, e, g). The total for all limbs is 12 additions and 36 permutations, and the large "bubble" of permutations ends up forcing many idle cycles for the addition unit. This occurs four times in each ladder step.

4.2. Changing the input/output format. There are two obvious sources of inefficiency in the computation described above. First, we need a final permutation to convert $(f-h, f+h, e-g, e+g)$ into $(f+h, f-h, e+g, e-g)$. Second, the middle permutation of (e, f, g, h) into (f, h, e, g) costs three permutation instructions, whereas (g, h, e, f) would cost only one.

The first problem arises from a tension between Intel's `vaddsubpd`, which always subtracts in the first position, and the definition of H, which always adds in the first position. A simple way to resolve this tension is to store (t, z, y, x) instead of (x, y, z, t) for the input, and (t', z', y', x') instead of (x', y', z', t') for the output; the final permutation then naturally disappears. It is easy to adjust the other permutations accordingly, along with constants such as $(1, a^2/b^2, a^2/c^2, a^2/d^2)$.

However, this does nothing to address the second problem. Different permutations of (x, y, z, t) as input and output end up requiring different middle permutations, but these middle permutations are never exactly the left-right swap provided by `vperm2f128`.

We do better by generalizing the input/output format to allow negations. For example, if we start with $(x, -y, z, t)$, permute into $(-y, x, t, z)$, and apply `vaddsubpd`, we obtain $(x+y, x-y, z-t, t+z)$. Observe that this is not the same as the $(x-y, x+y, z-t, t+z)$ that we obtained earlier: the first two entries have been exchanged.

It turns out to be best to negate z, i.e., to start from $(x, y, -z, t)$. Then `vpermilpd` gives $(y, x, t, -z)$, and `vaddsubpd` gives $(x-y, x+y, -z-t, t-z)$, which we now abbreviate as (e, f, g, h). Next `vperm2f128` gives (g, h, e, f), and independently `vpermilpd` gives (f, e, h, g). Finally, `vaddsubpd` gives $(f-g, h+e, h-e, f+g)$. This is exactly $(x', t', -z', y')$ where $(x', y', z', t') = H(x, y, z, t)$.

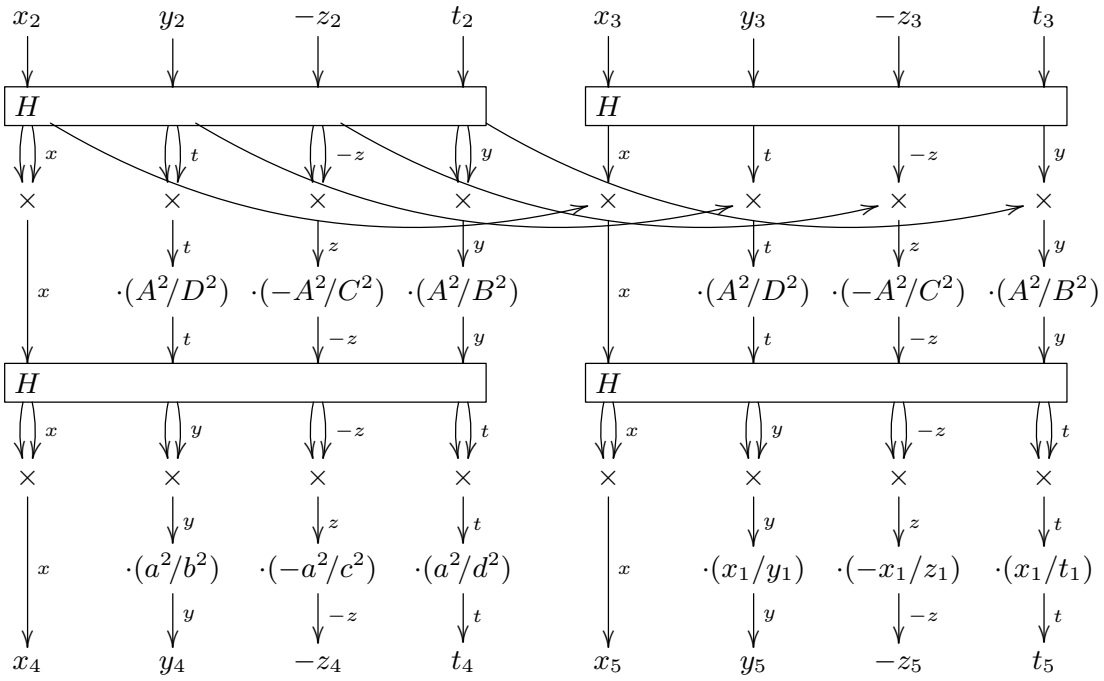


Fig. 4.3. Output format that we use for each operation in the right side of Figure 2.4 on Sandy Bridge, including permutations and negations to accelerate H.

The output format here is not the same as the input format: the positions of t and y have been exchanged. Fortunately, Figure 2.4 is partitioned by the H rows into two separate universes, and there is no need for the universes to use the same format. We use the $(x, y, -z, t)$ format at the top and bottom, and the $(x, t, -z, y)$ format between the two H rows. It is easy to see that exactly the same sequence of instructions works for all the copies of H, either producing $(x, y, -z, t)$ format from $(x, t, -z, y)$ format or vice versa.

$\mathbf{S^4}$ and $\mathbf{M^4}$ do not preserve negations: in effect, they switch from $(x, t, -z, y)$ format to (x, t, z, y) format. This is not a big problem, since we can reinsert the negation at any moment using a single multiplication or low-latency logic instruction (floating-point numbers use a sign bit rather than twos-complement, so negation is simply xor with a 1 in the sign bit). Even better, in Figure 2.4(b), the problem disappears entirely: each $\mathbf{S^4}$ and $\mathbf{M^4}$ is followed immediately by a constant multiplication, and so we simply negate the appropriate constants. The resulting sequence of formats is summarized in Figure 4.3.

Each H now costs 12 additions and just 18 permutations. The number of non-addition cycles that need to be overlapped with operations before and after H has dropped from the original 24 to just 6.

4.4. Exploiting double precision. We gain a further factor of 2 by temporarily converting from radix $2^{127/6}$ to radix $2^{127/3}$ during the computation of H. This means that, just before starting H, we replace the six limbs $(h_0, h_1, h_2, h_3, h_4, h_5)$ representing $h_0 + h_1 + h_2 + h_3 + h_4 + h_5$ by three limbs $(h_0 + h_1, h_2 + h_3, h_4 + h_5)$.

These three sums, and the intermediate H results, still fit into double-precision floating-point numbers.

It is essential to switch each output integer back to radix $2^{127/6}$ so that each output limb is small enough for the subsequent multiplication. Converting three limbs into six is slightly less expensive than three carries; in fact, converting from six to three and back to six uses exactly the same operations as three carries, although in a different order.

We further reduce the conversion cost by the following observation. Except for the $\mathbf{M^4}$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$, each of our multiplication results uses six carries, as explained in Section 3.4. However, if we are about to add h_0 to h_1 for input to H, then there is no reason to carry $h_0 \to h_1$, so we simply skip that carry; we similarly skip $h_2 \to h_3$ and $h_4 \to h_5$. These skipped carries exactly cancel the conversion cost.

For the $\mathbf{M^4}$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ the analysis is different: h_0 is large enough to affect h_2, and if we skipped carrying $h_0 \to h_1 \to h_2$ then the output of H would no longer be safe as input to a subsequent multiplication. We thus carry $h_0 \to h_1$, $h_2 \to h_3$, and $h_4 \to h_5$ in parallel; and then $h_1 \to h_2$, $h_3 \to h_4$, and $h_5 \to h_0$ in parallel. In effect this $\mathbf{M^4}$ uses 9 carries, counting the cost of conversion, whereas in Section 3.4 it used only 8.

To summarize, all of these conversions for all four H cost just one extra carry, while reducing 48 additions and 72 permutations to 24 additions and 36 permutations.

4.5. Conditional swaps. A ladder step starts from an input $(X(nP), X((n+1)P))$, which we abbreviate as $L(n)$, and produces $L(2n)$ as output. Swapping the two halves of the input, applying the same ladder step, and swapping the two halves of the output produces $L(2n+1)$ instead; one way to see this is to observe that $L(-n-1)$ is exactly the swap of $L(n)$.

Consequently one can reach $L(2n+\epsilon)$ for $\epsilon \in \{0,1\}$ by starting from $L(n)$, conditionally swapping, applying the ladder step, and conditionally swapping again, where the condition bit is exactly ϵ. A standard ladder reaches $L(n)$ by applying this idea recursively. A standard *constant-time* ladder reaches $L(n)$ by applying this idea for exactly ℓ steps, starting from $L(0)$, where n is known in advance to be between 0 and $2^\ell - 1$. An alternate approach is to first add to n an appropriate multiple of the order of P, producing an integer known to be between (e.g.) $2^{\ell+1}$ and $2^{\ell+2} - 1$, and then start from $L(1)$. We use a standard optimization, merging the conditional swap after a ladder step into the conditional swap before the next ladder step, so that there are just $\ell + 1$ conditional swaps rather than 2ℓ.

One way to conditionally swap field elements x and x' using floating-point arithmetic is to replace (x, x') with $(x + b(x' - x), x' - b(x' - x))$ where b is the condition bit, either 0 or 1. This takes three additions and one multiplication (times 6 limbs, times 4 field elements to swap). It is better to use logic instructions: replace each addition with `xor`, replace each multiplication with `and`, and replace b with an all-1 or all-0 mask computed from b. On the Sandy Bridge,

logic instructions have low latency and are handled by the permutation unit, which is much less of a bottleneck for us than the addition unit.

We further improve the performance of the conditional swap as follows. The $\mathbf{M^4}$ on the right side of Figure 4.3 is multiplying H of the left input by H of the right input. This is commutative: it does not depend on whether the inputs are swapped. We therefore put the conditional swap *after* the first row of H computations, and multiply the H outputs directly, rather than multiplying the swap outputs. This trick has several minor effects and one important effect.

A minor advantage is that this trick removes all use of the right half of the swap output; i.e., it replaces the conditional swap with a conditional move. This reduces the original 24 logic instructions to just 18.

Another minor advantage is as follows. The Sandy Bridge has a vectorized conditional-select instruction `vblendvpd`. This instruction occupies the permutation unit for 2 cycles, so it is no better than the 4 traditional logic instructions for a conditional swap: a conditional swap requires two conditional selects. However, this instruction *is* better than the 3 traditional logic instructions for a conditional move: a conditional move requires only one conditional select. This replaces the original logic instructions with 6 conditional-select instructions, consuming just 12 cycles.

A minor disadvantage is that the first $\mathbf{M^4}$ and $\mathbf{S^4}$ are no longer able to share precomputations of multiplications by 2^{-127}. This costs us 3 multiplication instructions.

The important effect is that this trick reduces latency, allowing the $\mathbf{M^4}$ to start much sooner. Adding this trick immediately produced a 5% reduction in our cycle counts.

4.6. Total operations. We treat Figure 2.4(b) as $2\mathbf{M^4} + 3\mathbf{S^4} + 3\mathbf{m^4} + 4H$.

The main computations of h_i, not counting precomputations and carries, cost 30 additions and 36 multiplications for each $\mathbf{M^4}$, 15 additions and 21 multiplications for each $\mathbf{S^4}$, and 0 additions and 6 multiplications for each $\mathbf{m^4}$. The total here is 105 additions and 153 multiplications.

The $\mathbf{M^4}$ by $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ allows precomputations outside the loop. The other $\mathbf{M^4}$ consumes 5 multiplications for precomputations, and each $\mathbf{S^4}$ consumes 8 multiplications for precomputations; the total here is 29 multiplications. We had originally saved a few multiplications by sharing precomputations between the first $\mathbf{S^4}$ and the first $\mathbf{M^4}$, but this is incompatible with the more important trick described in Section 4.5.

There are a total of 24 additions in the four H, as explained in Section 4.4. There are also 51 carries (counting the conversions described in Section 4.4 as carries), each consuming 3 additions and 2 multiplications, for a total of 153 additions and 102 multiplications.

The grand total is 282 additions and 284 multiplications, evidently requiring at least 284 cycles for each iteration of the main loop. Recall that there are various options to trade multiplications for additions: each $\mathbf{S^4}$ has 5 precomputed doublings that can each be converted from 1 multiplication to 1 addition, and each carry can be converted from 3 additions and 2 multiplications to 4 additions

and 0 multiplications (or 4 additions and 1 multiplication for $h_5 \to h_0$). We could use either of these options to eliminate one multiplication, reducing the 284-cycle lower bound to 283 cycles, but to reduce latency we ended up instead using the first option to eliminate 10 multiplications and the second option to eliminate 35 multiplications, obtaining a final total of 310 additions and 239 multiplications. These totals have been computer-verified.

We wrote functions in assembly for $\mathbf{M^4}$, $\mathbf{S^4}$, etc., but were still over 500 cycles. Given the Sandy Bridge floating-point latencies, and the requirement to keep *two* floating-point units constantly busy, we were already expecting instruction scheduling to be much more of an issue for this software than for typical integer-arithmetic software. We used various standard optimization techniques that were already used in several previous DH speed records: we merged the functions into a single loop, reorganized many computations to save registers, and eliminated many loads and stores. After building a new Sandy Bridge simulator and experimenting with different instruction schedules we ended up with our current loop, just 338 cycles, and a total of 88916 Sandy Bridge cycles for scalar multiplication. The main loop explains 84838 of these cycles; the remaining cycles are spent outside the ladder, mostly on converting $(x : y : z : t)$ to $(x/y : x/z : x/t)$ for output.

5 Cortex-A8

The low-power ARM Cortex-A8 core is the CPU core in the iPad 1, iPhone 4, Samsung Galaxy S, Motorola Droid X, Amazon Kindle 4, etc. Today a Cortex-A8 CPU, the Allwinner A10, costs just $5 in bulk and is widely used in low-cost tablets, set-top boxes, etc. Like Sandy Bridge, Cortex-A8 is not the most recent microarchitecture, but its very wide deployment and use make it a sensible choice of platform for optimization and performance comparisons.

Bernstein and Schwabe in [15] (CHES 2012) analyzed the vector capabilities of the Cortex-A8 for various cryptographic primitives, and in particular set a new speed record for high-security DH, namely 460200 Cortex-A8 cycles. We do much better, just 274593 Cortex-A8 cycles, measured on a Freescale i.MX515. Our basic vectorization approach is the same for Cortex-A8 as for Sandy Bridge, and many techniques are reused, but there are also many differences. The rest of this section explains the details.

5.1. Cortex-A8 vector units. Each Cortex-A8 core has two 128-bit vector units operating in parallel on vectors of four 32-bit integers or two 64-bit integers:

- The arithmetic port takes one cycle for vector addition, with latency 2; or two cycles for vector multiplication (two 64-bit products ac, bd given 32-bit inputs a, b and c, d), with latency 7. Logic operations also use the arithmetic port.
- The load/store port handles loads, stores, and permutations. ARM's Cortex-A8 documentation [5] indicates that the load/store port can carry out one

128-bit load every cycle. Beware, however, that there are throughput limits on the L1 cache. We have found experimentally that the common TI Sitara Cortex-A8 CPU (used, e.g., in the Beaglebone Black development board) needs three cycles from one load until the next (this is what we call "Cortex-A8-slow"), while other Cortex-A8 CPUs ("Cortex-A8-fast") can handle seven consecutive cycles of loads without penalty.

There are three obvious reasons for Cortex-A8 cycle counts to be much larger than Sandy Bridge cycle counts: registers are only 128 bits, not 256 bits; there are only 2 ports, not 6; and multiplication throughput is 1 every 2 cycles, not 1 every cycle. However, there are also speedups on Cortex-A8. There is (as in Haswell's floating-point units—see Appendix B) a vector multiply-accumulate instruction with the same throughput as vector multiplication. A sequence of m consecutive multiply-accumulate instructions that all accumulate into the same register executes in $2m$ cycles (unlike Haswell), effectively reducing multiplication latency from 7 to 1. Furthermore, Cortex-A8 multiplication produces 64-bit integer products, while Sandy Bridge gives only 53-bit-mantissa products.

5.2. Representation. We decompose an integer f modulo $2^{127}-1$ into *five* integer pieces in radix $2^{127/5}$: i.e., we write f as $f_0+2^{26}f_1+2^{51}f_2+2^{77}f_3+2^{102}f_4$. Compared to Sandy Bridge, having 20% more room in 64-bit integers than in 53-bit floating-point mantissas allows us to reduce the number of limbs from 6 to 5. We require the small integers f_0, f_1, f_2, f_3, f_4 to be *unsigned* because this reduces carry cost from 4 integer instructions to 3.

We arrange four integers x, y, z, t modulo $2^{127}-1$ in five 128-bit vectors: (x_0, y_0, x_1, y_1); (x_2, y_2, x_3, y_3); (x_4, y_4, z_4, t_4); (z_0, t_0, z_1, t_1); (z_2, t_2, z_3, t_3). This representation is designed to minimize permutations in **M**, **S**, and H. For example, computing $(x_0+z_0, y_0+t_0, x_1+z_1, y_1+t_1)$ takes just one addition without any permutations. The Cortex-A8 multiplications take two pairs of inputs at a time, rather than four as on Sandy Bridge, so there is little motivation to put (x_0, y_0, z_0, t_0) into a vector.

5.3. Optimizing M. Given an integer f as above and an integer $g = g_0 + 2^{26}g_1 + 2^{51}g_2 + 2^{77}g_3 + 2^{102}g_4$, the product fg modulo $2^{127}-1$ is $h = h_0 + 2^{26}h_1 + 2^{51}h_2 + 2^{77}h_3 + 2^{102}h_4$, with

$$\begin{aligned}
h_0 &= f_0g_0 + 2f_1g_4 + 2f_2g_3 + 2f_3g_2 + 2f_4g_1,\\
h_1 &= f_0g_1 + \ f_1g_0 + \ f_2g_4 + 2f_3g_3 + \ f_4g_2,\\
h_2 &= f_0g_2 + 2f_1g_1 + \ f_2g_0 + 2f_3g_4 + 2f_4g_3,\\
h_3 &= f_0g_3 + \ f_1g_2 + \ f_2g_1 + \ f_3g_0 + \ f_4g_4,\\
h_4 &= f_0g_4 + 2f_1g_3 + \ f_2g_2 + 2f_3g_1 + \ f_4g_0.
\end{aligned}$$

There are 25 multiplications f_ig_j; additions are free as part of multiply-accumulate instructions. We precompute $2f_1, 2f_2, 2f_3, 2f_4$ so that these values can be reused for another multiplication. These precomputations can be done by using either 4 shift or 4 addition instructions. Both shift and addition use 1

cycle per instruction, but addition has a lower latency. See Section 5.6 for the cost of carries.

5.4. Optimizing S. The idea of optimizing **S** in Cortex-A8 is quite similar to Sandy Bridge; for details see Section 3.3. We state here only the operation count. Besides precomputation and carry, we use 15 multiplication instructions; some of those are actually multiply-accumulate instructions. From now on we describe both multiplication instructions and multiply-accumulate instructions as "multiplications" without further comment.

5.5. Optimizing m. For **m** we compute only $h_0 = cf_0$, $h_1 = cf_1$, $h_2 = cf_2$, $h_3 = cf_3$, and $h_4 = cf_4$, again exploiting the small constants stated in Section 2.6.

Recall that we use *unsigned* representation. We always multiply absolute values, then negate results as necessary by subtracting from $2^{129} - 4$: $n_0 = 2^{28}-4-h_0$, $n_1 = 2^{27}-4-h_1$, $n_2 = 2^{28}-4-h_2$, $n_3 = 2^{27}-4-h_3$, $n_4 = 2^{27}-4-h_4$.

Negating any subsequence of x, y, z, t costs at most 5 vector subtractions. Negating only x or y, or both x and y, costs only 3 subtractions, because our representation keeps x, y within 3 vectors. The same comment applies to z and t. The specific **m** in Section 2.6 end up requiring a total of 13 subtractions with the same cost as 13 additions.

5.6. Carries. Each multiplication uses at worst 6 serial carries $h_1 \to h_2 \to h_3 \to h_4 \to h_0 \to h_1$, each costing 3 additions. Various carries are eliminated by the ideas of Section 3.4.

5.7. Hadamard transform. See Appendix A.

5.8. Total arithmetic. We view Figure 2.4(b) as $4\mathbf{M^2} + 6\mathbf{S^2} + 6\mathbf{m^2} + 4H$. Here we combine x multiplications and y multiplications into a vectorized $\mathbf{M^2}$, and similarly combine z multiplications and t multiplications; this fits well with the Cortex-A8 vector multiplication instruction, which outputs two products.

The main computations of h_i, not counting precomputations and carries, cost 0 additions and 25 multiplications for each **M**, 0 additions and 15 multiplications for each **S**, 0 additions and 5 multiplications for each **m**, and 15 additions for each H block. The total here is 60 additions and 220 multiplications.

Each **M** costs 4 additions for precomputations, and each **S** also costs 4 additions for precomputations. Some precomputations can be reused. The cost of precomputations is 20 additions.

There are 10 carry blocks using 6 carries each, and 6 carry blocks using 5 carries each. Each carry consists of 1 shift, 1 addition, and 1 logical `and`. This cost is equivalent to 3 additions. There are another 13 additions needed to handle negation. Overall the carries cost 283 additions. Two conditional swaps, each costing 9 additions, sum up to 18 additions.

In total we have 381 additions and 220 multiplications in our inner loop. This means that the inner loop takes at least 821 cycles.

We scheduled instructions carefully but ended up with some overhead beyond arithmetic: even though the arithmetic and the load/store unit can operate in parallel, latencies and the limited number of registers leave the arithmetic unit

idle for some cycles. Sobole's simulator at [48], which we found very helpful, reports 966 cycles. Actual measurements report 986 cycles; the 251 ladder steps thus account for 247486 of our 273349 cycles.

References

[1] — (no editor), *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19–22, 2013*, IEEE Computer Society, 2013. ISBN 978-1-4673-6166-8. See [4].

[2] — (no editor), *Proceedings of the 23rd USENIX security symposium, August 20–22, 2014, San Diego, CA, USA*, USENIX, 2014. See [20], [51].

[3] Onur Acıiçmez, Billy Bob Brumley, Philipp Grabher, *New results on instruction cache attacks*, in CHES 2010 [41] (2010), 110–124. URL: http://www.iacr.org/archive/ches2010/62250105/62250105.pdf. Citations in this document: §1.2.

[4] Nadhem J. AlFardan, Kenneth G. Paterson, *Lucky Thirteen: breaking the TLS and DTLS record protocols*, in S&P 2013 [1] (2013), 526–540. URL: http://www.isg.rhul.ac.uk/tls/TLStiming.pdf. Citations in this document: §1.2.

[5] ARM Limited, *Cortex-A8 technical reference manual, revision r3p2*, 2010. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/DDI0344K_cortex_a8_r3p2_trm.pdf. Citations in this document: §5.1.

[6] Vijay Atluri, Claudia Daz (editors), *Computer security — ESORICS 2011 — 16th European symposium on research in computer security, Leuven, Belgium, September 12–14, 2011, proceedings*, Lecture Notes in Computer Science, 6879, Springer, 2011. ISBN 978-3-642-23821-5. See [19].

[7] Josh Benaloh (editor), *Topics in cryptology — CT-RSA 2014 — the cryptographer's track at the RSA Conference 2014, San Francisco, CA, USA, February 25–28, 2014, proceedings*, Lecture Notes in Computer Science, 8366, Springer, 2014. ISBN 978-3-319-04851-2. See [24].

[8] Daniel J. Bernstein, *Floating-point arithmetic and message authentication* (2004). URL: http://cr.yp.to/papers.html#hash127. Citations in this document: §3, §3.1, §3.4, §B.2.

[9] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [52] (2006), 207–228. URL: http://cr.yp.to/papers.html#curve25519. Citations in this document: §1, §1.2, §3.1, §D.2.

[10] Daniel J. Bernstein, *Elliptic vs. hyperelliptic, part 1* (2006). URL: http://cr.yp.to/talks.html#2006.09.20. Citations in this document: §2.2, §2.2, §2.1, §2.3, §2.3, §2.5, §2.5, §3, §4.

[11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, in CHES 2011 [46] (2011), 124–142; see also newer version [12]. URL: https://eprint.iacr.org/2011/368. Citations in this document: §1, §1.4, §1.4, §1.4, §1.4, §D.2.

[12] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*, Journal of Cryptographic Engineering **2** (2012), 77–89; see also older version [11]. URL: https://eprint.iacr.org/2011/368.

[13] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 25 September 2014 (2014). URL: http://bench.cr.yp.to. Citations in this document: §1.

[14] Daniel J. Bernstein, Tanja Lange, *Hyper-and-elliptic-curve cryptography*, Special Issue A (Algorithmic Number Theory Symposium XI), LMS Journal of Computation and Mathematics **17** (2014), 181–202. URL: http://dx.doi.org/10.1112/S1461157014000394. Citations in this document: §D.1, §D.1.

[15] Daniel J. Bernstein, Peter Schwabe, *NEON crypto*, in CHES 2012 [**47**] (2012), 320–339. URL: http://cr.yp.to/papers.html#neoncrypto. Citations in this document: §1.4, §1.4, §1.4, §3, §5.

[16] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems — CHES 2013 — 15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [18], [44].

[17] Joppe W. Bos, Craig Costello, Huseyin Hisil, Kristin Lauter, *Fast cryptography in genus 2*, in Eurocrypt 2013 [**36**] (2013), 194–210. URL: https://eprint.iacr.org/2012/670. Citations in this document: §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §1.2, §1.2, §1.2, §1.2, §1.2, §1.2, §1.4, §1.4, §1.4, §1.4, §1.4, §2.6, §2.6, §D.1, §D.2, §D.2.

[18] Joppe W. Bos, Craig Costello, Huseyin Hisil, Kristin Lauter, *High-performance scalar multiplication using 8-dimensional GLV/GLS decomposition*, in CHES 2013 [**16**] (2013), 331–348. URL: https://eprint.iacr.org/2013/146. Citations in this document: §1, §1, §1.4.

[19] Billy Bob Brumley, Nicola Tuveri, *Remote timing attacks are still practical*, in ESORICS 2011 [**6**] (2011), 355-371. URL: https://eprint.iacr.org/2011/232. Citations in this document: §1.2.

[20] Stephen Checkoway, Matthew Fredrikson, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, *On the practical exploitability of Dual EC in TLS implementations*, in USENIX Security 2014 [**2**] (2014). URL: https://projectbullrun.org/dual-ec/index.html. Citations in this document: §D.2.

[21] David V. Chudnovsky, Gregory V. Chudnovsky, *Sequences of numbers generated by addition in formal groups and new primality and factorization tests*, Advances in Applied Mathematics **7** (1986), 385–434. MR 88h:11094. URL: http://dx.doi.org/10.1016/0196-8858(86)90023-0. Citations in this document: §2.1.

[22] Craig Costello, Huseyin Hisil, Benjamin Smith, *Faster compact Diffie–Hellman: endomorphisms on the x-line*, in Eurocrypt 2014 [**43**] (2014), 183–200. URL: https://eprint.iacr.org/2013/692. Citations in this document: §1, §1.2, §1.2, §1.2, §1.4, §1.4, §D.2.

[23] Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine*, in Africacrypt 2009 [**45**] (2009), 368–385. URL: http://cryptojedi.org/users/peter/#celldh. Citations in this document: §3.

[24] Armando Faz-Hernández, Patrick Longa, Ana H. Sánchez, *Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves*, in CT-RSA 2014 [**7**] (2013), 1–27. URL: https://eprint.iacr.org/2013/158. Citations in this document: §1, §1, §1, §1.2, §1.4, §1.4, §1.4, §1.4, §1.4.

[25] Agner Fog, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs* (2014). URL: http://agner.org/optimize/. Citations in this document: §3.1.

[26] Steven Galbraith, Xibin Lin, Michael Scott, *Endomorphisms for faster elliptic curve cryptography on a large class of curves*, in Eurocrypt 2009 [**37**] (2009), 518–535. URL: https://eprint.iacr.org/2008/194. Citations in this document: §1.

[27] Robert P. Gallant, Robert J. Lambert, Scott A. Vanstone, *Faster point multiplication on elliptic curves with efficient endomorphisms*, in Crypto 2001 [**38**] (2001), 190–200. MR 2003h:14043. URL: http://www.iacr.org/archive/crypto2001/21390189.pdf. Citations in this document: §1.

[28] Robert P. Gallant, Robert J. Lambert, Scott A. Vanstone, *U.S. patent 7110538: method for accelerating cryptographic operations on elliptic curves* (2006). URL: http://www.freepatentsonline.com/7110538.html. Citations in this document: §1.

[29] Robert P. Gallant, Robert J. Lambert, Scott A. Vanstone, *U.S. patent 7995752: method for accelerating cryptographic operations on elliptic curves* (2011). URL: http://www.freepatentsonline.com/7995752.html. Citations in this document: §1.

[30] Pierrick Gaudry, *Variants of the Montgomery form based on Theta functions* (2006); see also newer version [**31**]. URL: http://www.loria.fr/~gaudry/publis/toronto.pdf. Citations in this document: §2.2, §2.2, §2.1.

[31] Pierrick Gaudry, *Fast genus 2 arithmetic based on Theta functions*, Journal of Mathematical Cryptology **1** (2007), 243–265; see also older version [**30**]. URL: http://webloria.loria.fr/~gaudry/publis/arithKsurf.pdf. Citations in this document: §2.1.

[32] Pierrick Gaudry, David Lubicz, *The arithmetic of characteristic 2 Kummer surfaces and of elliptic Kummer lines*, Finite Fields and Their Applications **15** (2009), 246–260. URL: http://hal.inria.fr/inria-00266565/PDF/c2.pdf. Citations in this document: §2.5, §B.1.

[33] Pierrick Gaudry, Éric Schost, *Genus 2 point counting over prime fields*, Journal of Symbolic Computation **47** (2012), 368–400. URL: http://www.csd.uwo.ca/~eschost/publications/countg2.pdf. Citations in this document: §2.6.

[34] Mike Hamburg, *Fast and compact elliptic-curve cryptography* (2012). URL: https://eprint.iacr.org/2012/309. Citations in this document: §1, §1.4, §1.4.

[35] Huseyin Hisil, Craig Costello, *Jacobian coordinates on genus 2 curves* (2014). URL: https://eprint.iacr.org/2014/385. Citations in this document: §D.1.

[36] Thomas Johansson, Phong Q. Nguyen (editors), *Advances in cryptology — EUROCRYPT 2013, 32nd annual international conference on the theory and applications of cryptographic techniques, Athens, Greece, May 26–30, 2013, proceedings*, Lecture Notes in Computer Science, 7881, Springer, 2013. ISBN 978-3-642-38347-2. See [17].

[37] Antoine Joux (editor), *Advances in cryptology — EUROCRYPT 2009, 28th annual international conference on the theory and applications of cryptographic techniques, Cologne, Germany, April 26–30, 2009, proceedings*, Lecture Notes in Computer Science, 5479, Springer, 2009. ISBN 978-3-642-01000-2. See [26].

[38] Joe Kilian (editor), *Advances in cryptology — CRYPTO 2001, 21st annual international cryptology conference, Santa Barbara, California, USA, August 19–23, 2001, proceedings*, Lecture Notes in Computer Science, 2139, Springer, 2001. ISBN 3-540-42456-3. MR 2003d:94002. See [27].

[39] Donald Knuth, *Structured programming with go to statements*, Computing Surveys **6** (1974), 261–301. Citations in this document: §D.

[40] Patrick Longa, Francesco Sica, *Four-dimensional Gallant–Lambert–Vanstone scalar multiplication*, in Asiacrypt 2012 [**50**] (2012), 718–739. URL: https://eprint.iacr.org/2011/608. Citations in this document: §1, §1, §1.4, §1.4.

[41] Stefan Mangard, François-Xavier Standaert (editors), *Cryptographic hardware and embedded systems, CHES 2010, 12th international workshop, Santa Barbara,*

CA, USA, August 17–20, 2010, proceedings, Lecture Notes in Computer Science, 6225, Springer, 2010. ISBN 978-3-642-15030-2. See [3].

[42] Peter L. Montgomery, *Speeding the Pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), 243–264. ISSN 0025-5718. MR 88e:11130. URL: http://links.jstor.org/sici?sici=0025-5718(198701)48:177<243:STPAEC>2.0.CO;2-3. Citations in this document: §3.

[43] Phong Q. Nguyen, Elisabeth Oswald (editors), *Advances in cryptology — EUROCRYPT 2014 — 33rd annual international conference on the theory and applications of cryptographic techniques, Copenhagen, Denmark, May 11–15, 2014, proceedings*, Lecture Notes in Computer Science, 8441, Springer, 2014. ISBN 978-3-642-55219-9. See [22].

[44] Thomaz Oliveira, Julio López, Diego F. Aranha, Francisco Rodríguez-Henríquez, *Lambda coordinates for binary elliptic curves*, in CHES 2013 [**16**] (2013), 311–330. URL: https://eprint.iacr.org/2013/131. Citations in this document: §1, §1, §1, §1.2, §1.4, §1.4, §1.4, §1.4, §1.4, §1.4, §B.1, §B.1, §B.1.

[45] Bart Preneel (editor), *Progress in cryptology — AFRICACRYPT 2009, second international conference on cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009, proceedings*, Lecture Notes in Computer Science, 5580, Springer, 2009. See [23].

[46] Bart Preneel, Tsuyoshi Takagi (editors), *Cryptographic hardware and embedded systems — CHES 2011, 13th international workshop, Nara, Japan, September 28–October 1, 2011, proceedings*, Lecture Notes in Computer Science, 6917, Springer, 2011. ISBN 978-3-642-23950-2. See [11].

[47] Emmanuel Prouff, Patrick Schaumont (editors), *Cryptographic hardware and embedded systems — CHES 2012 — 14th international workshop, Leuven, Belgium, September 9–12, 2012, proceedings*, Lecture Notes in Computer Science, 7428, Springer, 2012. ISBN 978-3-642-33026-1. See [15].

[48] Étienne Sobole, *Calculateur de cycle pour le Cortex A8* (2012). URL: http://pulsar.webshaker.net/ccc/index.php. Citations in this document: §5.8.

[49] Eran Tromer, Dag Arne Osvik, Adi Shamir, *Efficient cache attacks on AES, and countermeasures*, Journal of Cryptology **23** (2010), 37–71. URL: http://people.csail.mit.edu/tromer/papers/cache-joc-official.pdf. Citations in this document: §1.2.

[50] Xiaoyun Wang, Kazue Sako (editors), *Advances in cryptology — ASIACRYPT 2012, 18th international conference on the theory and application of cryptology and information security, Beijing, China, December 2–6, 2012, proceedings*, Lecture Notes in Computer Science, 7658, Springer, 2012. ISBN 978-3-642-34960-7. See [40].

[51] Yuval Yarom, Katrina Falkner, *Flush+Reload: a high resolution, low noise, L3 cache side-channel attack*, in USENIX Security 2014 [**2**] (2014). URL: https://eprint.iacr.org/2013/448. Citations in this document: §1.2, §1.2.

[52] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography — 9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [9].

A Cortex-A8 Hadamard transform

This appendix explains the details of how we compute H on the Cortex-A8.

Each of the 8 field additions in H requires 5 additions of limbs. One ladder step has four H, for a total of 160 limb additions, i.e., at least 40 vector additions.

Four of the field additions in H are actually subtractions. We handle subtractions by the same strategy as Section 5.6. The extra constants cost another 5 vector additions per H.

The detailed sequence of operations that we use on the Cortex-A8 is as follows. The Hadamard transform receives as input

$$\begin{aligned} r_0 &= (x_0, y_0, x_1, y_1); \\ r_1 &= (x_2, y_2, x_3, y_3); \\ r_2 &= (x_4, y_4, z_4, t_4); \\ r_3 &= (z_0, t_0, z_1, t_1); \\ r_4 &= (z_2, t_2, z_3, t_3). \end{aligned}$$

The output will be 5 registers $s_0, \ldots, s_4$ with

$$\begin{aligned} s_0 &= ((x_0+y_0)+(z_0+t_0), (x_0+y_0)-(z_0+t_0), (x_1+y_1)+(z_1+t_1), (x_1+y_1)-(z_1+t_1)), \\ s_1 &= ((x_0-y_0)+(z_0-t_0), (x_0-y_0)-(z_0-t_0), (x_1-y_1)+(z_1-t_1), (x_1-y_1)-(z_1-t_1)), \\ s_2 &= ((x_4+y_4)+(z_4+t_4), (x_4+y_4)-(z_4+t_4), (x_4-y_4)+(z_4-t_4), (x_4-y_4)-(z_4-t_4)), \\ s_3 &= ((x_2+y_2)+(z_2+t_2), (x_2+y_2)-(z_2+t_2), (x_3+y_3)+(z_3+t_3), (x_3+y_3)-(x_3+y_3)), \\ s_4 &= ((x_2-y_2)+(z_2-t_2), (x_2-y_2)-(z_2-t_2), (x_3-y_3)+(z_3-t_3), (x_3-y_3)-(x_3-y_3)). \end{aligned}$$

We begin with vector addition and subtraction to produce

$$\begin{aligned} t_0 &= (x_0 + z_0, y_0 + t_0, x_1 + z_1, y_1 + t_1), \\ t_1 &= (x_0 - z_0, y_0 - t_0, x_1 - z_1, y_1 - t_1), \\ t_2 &= (x_2 + z_2, y_2 + t_2, x_3 + z_3, y_3 + t_3), \\ t_3 &= (x_2 - z_2, y_2 - t_2, x_3 - z_3, y_3 - t_3). \end{aligned}$$

We would next like to add/subtract $x_0 + z_0$ with $y_0 + t_0$, also $x_1 + z_1$ with $y_1 + t_1$, and so on. Unfortunately, there are no instructions to add/subtract among or between left/right halves of vectors. There is a Cortex-A8 instruction `vtrn` which allows permuting two vectors (a, b, c, d) (e, f, g, h) to produce (a, e, c, g) (b, f, d, h), and can also permute two vectors (a, b, c, d) (e, f, g, h) to produce (a, b, c, g) (e, f, d, h). Another helpful instruction is `vswp` which swaps left and right halves of two vectors in various ways such as (a, b, c, d) (e, f, g, h) $\rightarrow$ (a, b, e, f) (c, d, g, h), and (a, b, c, d) (e, f, g, h) $\rightarrow$ (a, b, g, h) (e, f, c, d).

We apply `vtrn` to t_0 and t_1 to produce

$$\begin{aligned} t_4 &= (x_0 + z_0, x_0 - z_0, x_1 + z_1, x_1 - z_1), \\ t_5 &= (y_0 + t_0, y_0 - t_0, y_1 + t_1, y_1 - t_1). \end{aligned}$$

We then add and subtract to produce

$$\begin{aligned} t_6 &= (x_0 + z_0 + y_0 + t_0, x_0 - z_0 + y_0 - t_0, x_1 + z_1 + y_1 + t_1, x_1 - z_1 + y_1 - t_1), \\ t_7 &= (x_0 + z_0 - y_0 - t_0, x_0 - z_0 - y_0 + t_0, x_1 + z_1 - y_1 - t_1, x_1 - z_1 - y_1 + t_1). \end{aligned}$$

These are two of the desired output vectors from the Hadamard transform.

We could repeat similar steps for t_2 and t_3, but then there would be considerable overhead in handling the one remaining vector. To avoid arithmetic overhead we we permute three vectors together while performing arithmetic on two at a time. Specifically, we apply `vtrn` to t_2 and t_3 to produce

$$t_8 = (x_2 + z_2, x_2 - z_2, x_3 + z_3, x_3 - z_3),$$
$$t_9 = (y_2 + t_2, y_2 - t_2, y_3 + t_3, y_3 - t_3);$$

next use `vswp` (a, b, c, d) $(e, f, g, h) \to (a, b, e, f)$ (c, d, g, h) to t_8, r_2 to produce

$$t_{10} = (x_2 + z_2, x_2 - z_2, x_4, y_4),$$
$$t_{11} = (x_3 + z_3, x_3 - z_3, z_4, t_4);$$

and then use `vswp` (a, b, c, d) $(e, f, g, h) \to (a, b, g, h)$ (e, f, c, d) to t_9, t_{11} to produce

$$t_{12} = (y_2 + t_2, y_2 - t_2, z_4, t_4),$$
$$t_{13} = (x_3 + z_3, x_3 - z_3, y_3 + t_3, y_3 - t_3).$$

We then add and subtract t_{10} and t_{12} to produce

$$t_{14} = (x_2 + z_2 + y_2 + t_2, x_2 - z_2 + y_2 - t_2, x_4 + z_4, y_4 + t_4),$$
$$t_{15} = (x_2 + z_2 - y_2 - t_2, x_2 - z_2 - y_2 + t_2, x_4 - z_4, y_4 - t_4).$$

Next, we perform another sequence of permutations as follows: starting with using `vswp` (a, b, c, d) $(e, f, g, h) \to (e, f, c, d)$ (a, d, g, h) to t_{14} and t_{13} to produce

$$t_{16} = (x_3 + z_3, x_3 - z_3, x_4 + z_4, y_4 + t_4),$$
$$t_{17} = (x_2 + z_2 + y_2 + t_2, x_2 - z_2 + y_2 - t_2, y_3 + t_3, y_3 - t_3);$$

then using `vswp` (a, b, c, d) $(e, f, g, h) \to (a, b, e, f)$ (c, d, g, h) to t_{17} and t_{15} to produce

$$t_{18} = (x_2 + z_2 + y_2 + t_2, x_2 - z_2 + y_2 - t_2, x_2 + z_2 - y_2 - t_2, x_2 - z_2 - y_2 + t_2),$$
$$t_{19} = (y_3 + t_3, y_3 - t_3, x_4 - z_4, y_4 - t_4);$$

and then using `vtrn` (a, b, c, d) $(e, f, g, h) \to (a, b, c, g)$ (e, f, d, h) to t_{16} and t_{19} to produce

$$t_{20} = (x_3 + z_3, x_3 - z_3, x_4 + z_4, x_4 - z_4),$$
$$t_{21} = (y_3 + t_3, y_3 - t_3, y_4 + t_4, y_4 - t_4).$$

Now we are ready to add and subtract t_{20} with t_{21} to produce

$$t_{22} = (x_3 + z_3 + y_3 + t_3, x_3 - z_3 + y_3 - t_3, x_4 + z_4 + y_4 + t_4, x_4 - z_4 + y_4 - t_4),$$
$$t_{23} = (x_3 + z_3 - y_3 - t_3, x_3 - z_3 - y_3 + t_3, x_4 + z_4 - y_4 - t_4, x_4 - z_4 - y_4 + t_4).$$

Finally, we use `vswp` (a,b,c,d) $(e,f,g,h) \to (a,b,e,f)$ (c,d,g,h) to t_{22} and t_{23} to produce

$$t_{24} = (x_3 + z_3 + y_3 + t_3, x_3 - z_3 + y_3 - t_3, x_3 + z_3 - y_3 - t_3, x_3 - z_3 - y_3 + t_3),$$
$$t_{25} = (x_4 + z_4 + y_4 + t_4, x_4 - z_4 + y_4 - t_4, x_4 + z_4 - y_4 - t_4, x_4 - z_4 - y_4 + t_4);$$

and `vswp` (a,b,c,d) $(e,f,g,h) \to (a,b,e,f)$ (c,d,g,h) to t_{18} and t_{24} to produce

$$t_{26} = (x_2 + z_2 + y_2 + t_2, x_2 - z_2 + y_2 - t_2, x_3 + z_3 + y_3 + t_3, x_3 - z_3 + y_3 - t_3),$$
$$t_{27} = (x_2 + z_2 - y_2 - t_2, x_2 - z_2 - y_2 + t_2, x_3 + z_3 - y_3 - t_3, x_3 - z_3 - y_3 + t_3).$$

The vectors $t_6, t_7, t_{25}, t_{26}, t_{27}$ are the final results of the Hadamard transform.

B Haswell

Compared to Sandy Bridge (and Ivy Bridge), Haswell devotes considerably more transistors to multiplication. This appendix analyzes the impact of Haswell's new multipliers, and in particular explains how we achieved 54389 Haswell cycles for HECC.

B.1. Binary-polynomial multipliers. Haswell has an unusual level of hardware support for fast binary-polynomial multiplication. One should expect this to make binary-field ECC much faster on Haswell than on other CPUs, as illustrated by the impressive 55595 Haswell cycles in [44] for constant-time binary-field GLV+GLS ECC. (Note that recent advances in index-calculus discrete-logarithm algorithms for *multiplicative* groups of binary fields do not threaten the binary-field *curves* used in [44].)

Our HECC speeds are nevertheless faster. Perhaps there are further improvements in the approach of [44], but it is clear that HECC is an excellent cross-platform option and will continue to benefit from increased CPU support for vectorization, while it is not at all clear whether Intel will continue to expand the circuit area devoted to binary-field arithmetic, or whether other CPU manufacturers will follow. Note also that there are formulas in [32] for Kummer surfaces of binary hyperelliptic curves, opening up the possibility of a unification of these techniques.

B.2. Vectorized floating-point multipliers. Each Haswell core has *two* vectorized floating-point multiplication units: port 0 and port 1 each handle one 256-bit vector multiplication each cycle. Even better, these multipliers include integrated adders, so in one cycle a Haswell core can compute $ab + c$ and $de + f$, while in one cycle a Sandy Bridge core can compute at best ab and $d + f$. One should not think that this trivially reduces our cycle counts by a factor of 2:

- Multiplication latency is still 5 cycles, so to keep both multipliers busy one needs 10 independent multiplications. For comparison, keeping the Sandy Bridge multiplier busy needs only 5 independent multiplications.

- The integrated adders are useful only for additions (or subtractions) of products. There is no improvement to the performance of other addition instructions: each 256-bit vector addition consumes port 1 for one cycle. Obviously one can also take advantage of port 0 by rewriting $b + c$ as $1b + c$, but this still costs 1 arithmetic instruction rather than 0.5 arithmetic instructions, and it also increases latency from 3 to 5.

A further detail of importance for us is that the `vroundpd` instruction on the Haswell costs as much as two additions, whereas on Sandy Bridge it costs just one. We therefore switched to the carry approach from [8] described in Section 3.4. This reduced our starting 284 multiplications on Sandy Bridge to 190 multiplications: it eliminated 2 multiplications in each of 51 carries, except that the 8 carries $h_5 \to h_0$ (see Section 3.4) still require 1 multiplication each. Meanwhile this increased the original 282 additions to 333 additions.

Most of the additions in Sections 3.4 and 4 cannot be integrated into multiplications but the additions in Sections 3.2 and 3.3 naturally have the form "add ab into c". After integrating additions into multiplications we were left with 220 additions and 190 multiplications. These numbers have been computer-verified.

This arithmetic consumes at least 205 arithmetic cycles, i.e., at least 51455 cycles for 251 iterations of the main loop. The actual performance of our main loop at this point was much worse, 333 cycles. After initial rescheduling our eBACS-verified total cost was 72276 cycles; the main loop consumed 272 cycles, explaining 68272 cycles. We then switched to exploring a different approach described below.

B.3. Vectorized integer multipliers. Each Haswell core has one 4-way vectorized $32 \times 32 \to 64$ integer multiplier (port 0) and two 4-way vectorized 64-bit integer adders (port 5, competing with permutations, and port 1). Compared to floating-point vectors, Haswell's integer vectors have the obvious disadvantage of producing only 4 products per cycle instead of 8; on the other hand, integer vectors provide a noticeably better spread of operations across ports, the ability to efficiently use 5 limbs rather than 6, and a much lower addition latency.

We split each field element x into 5 unsigned limbs x_0, x_1, x_2, x_3, x_4; we use unsigned limbs here because Haswell's vectorized 64-bit right-shift instruction handles only unsigned integers. Each limb fits into 32 bits. We arrange four field elements (x, y, z, t) as five 256-bit vectors $(x_i, 0, y_i, 0, z_i, 0, t_i, 0)$, matching the input format used by the vectorized integer-multiplication instruction.

M involves 25 32-bit multiplications, 20 64-bit additions, and 6 carries, after a precomputation involving 4 32-bit doublings. Each carry consists of 1 shift, 1 addition, and 1 mask. For the precomputation we (1) use additions rather than shifts, because shift instructions compete with the multiplier on port 0, and (2) use 64-bit additions rather than 32-bit additions, because the extra efficiency of 32-bit additions is outweighed by the cost of extra permutations.

$\mathbf{M^4}$ thus uses 25 (vectorized) multiplication instructions, 26 or 30 addition instructions depending on whether precomputation is included, 6 shift instructions, and 6 mask instructions. Similarly, $\mathbf{S^4}$ uses 15 multiplication instructions, 20 addition instructions, 6 shift instructions, and 6 mask instructions; $\mathbf{m^4}$ uses

5 multiplication instructions, 5 addition instructions, 5 shift instructions, and 5 mask instructions. We use absolute values of all constants in $\mathbf{m}^4$; this means that we are implicitly negating x as output of $\mathbf{m}^4$, but we compensate for this in the H steps, as discussed in a moment.

We apply H to two inputs (x, y, z, t) and (x', y', z', t') at once. These two inputs actually consist of five vectors $(-x_i, 0, y_i, 0, z_i, 0, t_i, 0)$ and five vectors $(-x'_i, 0, y'_i, 0, z'_i, 0, t'_i, 0)$. For each i we build $(-x_i, -x'_i, y_i, y'_i, z_i, z'_i, t_i, t'_i)$ at the expense of a shift and a xor. (We could alternatively apply the same format conversions to four out of the five vectors in one H input, reducing the data flow between the left and right halves of Figure 2.4(b).) Undoing this format conversion at the end of H^2 might seem to involve a shift and a mask, but we actually skip the mask: each H output is used solely for multiplication, and the multiplication instructions implicitly mask their inputs.

The input to the core of H^2 is thus $(-x_i, -x'_i, y_i, y'_i, z_i, z'_i, t_i, t'_i)$, which for this paragraph we abbreviate as $(-x, y, z, t)$. For the first Hadamard level we obtain

- $(x-1, y, z, -t-1)$ by a xor of $(-x, y, z, t)$ with $(-1, 0, 0, -1)$, exploiting the twos-complement representation of integers;
- $(y, -x, t, z)$ by a permutation; and
- $(x+y-1, y-x, z+t, z-t-1)$ by an addition.

We then obtain $(x+y-1, z+t, y-x, z-t-1)$ by a (latency-3) permutation. For the second Hadamard level we obtain

- $(x+y-1, -z-t-1, x-y-1, z-t-1)$ by a xor with $(0, -1, -1, 0)$;
- $(z+t, x+y-1, z-t-1, y-x)$ by a permutation; and
- $(x+y+z+t-1, x+y-z-t-2, x-y+z-t-2, -(x-y-z+t)-1)$ by an addition.

A final addition of $(1, 2, 2, 1)$ would produce $(x+y+z+t, x+y-z-t, x-y+z-t, -(x-y-z+t))$. We actually add something larger (equal to $(1, 2, 2, 1)$ modulo $2^{255}-19$) to produce unsigned results, as in Section 5. The negation of $x-y-z+t$ disappears as in Figure 4.3.

Overall H^2 uses 11 vector instructions — 3 additions, 2 shifts, 3 permutations, and 3 logic instructions — for each of the 5 limbs. There are also 5 conditional-select instructions after the first H^2, as in Section 4.

We treat Figure 2.4(b) as $2\mathbf{M^4} + 3\mathbf{S^4} + 3\mathbf{m^4} + 2H^2$, for a total of 110 multiplications, 161 additions, 65 shifts, 75 logic instructions, and 30 permutations, plus the 5 conditional-select instructions, each of which is as expensive as 2 permutations. These totals have been computer-verified.

The 175 multiplications and shifts use port 0; the 161 additions use port 1; the $30 + 5 \cdot 2$ permutations use port 5; and the 75 logic instructions can use any of these ports. The most obvious bottleneck is 175 cycles for port 0. The most obvious scheduling challenge is to avoid having port 0 distracted by the logic operations. Our current software uses 197 cycles for the main loop.

C Lattice techniques

The maximum possible speedup from the following idea is small and as far as we can tell is outweighed by overhead, so we do not use it in our software, but we briefly describe the idea because it might be useful in other contexts.

One could scale $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ so that each limb is smaller, hopefully small enough to eliminate the need for carry chains in the relevant $\mathbf{M}$. There are 24 limbs (on Sandy Bridge) and approximately 2^{127} possible scalings, so one would expect a scaling to exist that makes all the limbs 5 bits smaller. However, finding this scaling appears to be a medium-dimensional lattice problem that would cost more to solve than it could possibly save. Scaling to four integers below 2^{96} would be a much easier lattice computation and would save the multiplications by top coefficients, but still does not appear to be worthwhile.

For comparison, scaling $(x_1 : y_1 : z_1 : t_1)$ to $(1 : x_1/y_1 : x_1/z_1 : x_1/t_1)$ is a one-dimensional lattice problem. The potential advantage of the higher-dimensional lattices in the previous paragraph is that they are compatible with our vectorization across the four coefficients.

D Fixed-base scalar multiplication

> *There is no doubt that the 'grail' of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We* should *forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
>
> —Donald E. Knuth, 1974 [39]

The simplest way to generate a DH key is to apply our variable-base-point scalar-multiplication software to the fixed base point

$$(x/y : x/z : x/t) = (6 : 1425141370032895085206830138138882335 76 : 1)$$

of prime order (the prime ending 339). The software takes constant time, and in particular takes the same time for DH key generation as it does for DH shared-secret computation.

D.1. Optimizing fixed-base scalar multiplication. The simplest approach is not the fastest approach. It is well known that fixed-base scalar multiplication, and in particular DH key generation, can be made much faster than variable-base scalar multiplication.

The standard way to speed up fixed-base scalar multiplication in genus 2 is to precompute various multiples of the base point on the Jacobian. For Jacobian addition formulas see, e.g., [17] and [35]. An alternate "hyper-and-elliptic" approach proposed recently in [14] is to carry out fixed-base scalar multiplication using precomputed points on an auxiliary elliptic curve, taking advantage of the

speed of elliptic-curve additions, and then map from the elliptic curve to the Jacobian; this does not work with the Kummer surface that we used, but it does work with other small-coefficient Kummer surfaces constructed in [14]. Mapping from Jacobian to Kummer takes a few dozen additional multiplications.

For memory-limited platforms there are still significant speedups from a small number of precomputed points. For example, precomputing just 4 points reduces the number of doublings by a factor of 4. When the number of precomputed points is sufficiently small we could also carry out lattice precomputations as in Appendix C to noticeably reduce the size of the projective representations of those points; alternatively, we could generate new base points meeting various size criteria.

D.2. Reusing DH keys. There are two major types of DH keys: *long-term* (or "static") DH keys used as traditional identifiers, and *ephemeral* DH keys that are erased to provide forward secrecy. For example, HTTPS supports certified long-term "ECDH" SSL keys assigned to web servers, and it also supports ephemeral "ECDHE" SSL keys that provide forward secrecy.

A long-term DH key involves just one fixed-base scalar multiplication (for key generation) and is then bottlenecked by variable-base scalar multiplications using public keys of other parties. In this context it is obvious that there is negligible benefit in speeding up fixed-base scalar multiplication.

The same optimization applies to ephemeral DH: an implementor who finds that key-generation time is a bottleneck can simply reuse ephemeral DH keys. If a DH key is reused just 1000 times, and key generation is implemented in the simplest way as variable-base scalar multiplication, then key generation is below 0.1% of the total DH cost. Of course, standard fixed-base speedups (see above) make DH key generation another few times faster, but this has negligible benefit. The critical speedup comes from key reuse.

As a concrete example, Microsoft's SSL library (SChannel) reuses ephemeral DH keys for 2 hours, according to [20, page 8]. Even if the reuse intervals were drastically shortened, and keys were discarded after just 2 seconds, the ephemeral key-generation time (a small fraction of a millisecond using ECC or HECC) would be completely unnoticeable.

These are not new observations: they are the reason that the cost of variable-base scalar multiplication is the primary metric used in the DH literature. For example, [9] ("new Diffie–Hellman speed records") mentions that fixed-base speedups exist and then says that these speedups are "negligible in the Diffie–Hellman context ... since each key is used many times". As another example, when [17] ("Fast cryptography") advertises a "new software speed record" for "constant-time scalar multiplication", it is referring to variable-base scalar multiplication; [17] says far less about fixed-base performance. As yet another example, when [22] ("Faster compact Diffie–Hellman") advertises "fast elliptic curve scalar multiplication, optimized for Diffie–Hellman Key Exchange", it is referring to variable-base scalar multiplication.

Of course, in contexts where it is important for every key to be erased as quickly as possible, or where any key reuse would give away valuable metadata,

one should switch to the simplest form of ephemeral DH. This means carrying out one fixed-base scalar multiplication to generate a single-use key, and one variable-base scalar multiplication to generate a shared secret. Even in this (arguably important) corner of the DH universe, fixed-base scalar multiplication (with the standard optimizations) consumes relatively little time, so making variable-base scalar multiplication $N\%$ faster is more important than making fixed-base scalar multiplication $N\%$ faster.

Fixed-base scalar multiplication is far more important outside the DH context: for example, it is the primary bottleneck in signature generation. See, e.g., [11]. We do not discuss this further: this paper focuses on DH.